eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Peter Pepper
Petra Hofstedt

Funktionale Programmierung

Sprachdesign und Programmiertechnik

Mit 57 Abbildungen und 18 Tabellen

Springer

Peter Pepper
Petra Hofstedt

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Franklinstraße 28/29
10587 Berlin
pepper@cs.tu-berlin.de
ph@cs.tu-berlin.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1614-5216
ISBN-10 3-540-20959-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-20959-1 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz: Druckfertige Daten der Autoren
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142 YL – 5 4 3 2 1 0

Für Claudia

Meinen Eltern
Christel und Klaus Hofstedt

Vorwort

Lernen ist wie Rudern gegen den Strom; sobald man damit aufhört, treibt man zurück.
Chinesische Weisheit

Auch wenn es sich bei diesem Text in gewissem Sinn um die Fortführung des einführenden Lehrbuchs [111], so ist sein Ziel doch ein ganz anderes. In dem Vorgängerbuch [111] geht es vor allem darum, dem Anfänger einen Einstieg in das konkrete Programmieren mit existierenden Funktionalen Sprachen zu geben. Dafür werden drei Sprachen benutzt: OPAL, ML und HASKELL.

Das vorliegende Buch wendet sich an fortgeschrittene Leser, die mehr über Funktionale Sprachen – und nicht nur diese – lernen wollen. Und das betrifft nicht nur real existierende Sprachen, sondern – was noch wichtiger ist – Konzepte, die heute untersucht, diskutiert und ausgearbeitet und in der einen oder anderen Weise ihren Weg in künftige Sprachen finden werden.

Der Weg, den wir dazu wählen, ist, eine fiktive Sprache zu behandeln, die sich zwar sehr stark an bekannten Sprachen – vor allem wieder an OPAL, ML und HASKELL – orientiert, aber in entscheidenden Punkten über sie hinaus geht und neue Ideen realisiert. Der Nachteil ist, dass es keinen Compiler gibt, den man aus dem Internet herunterladen könnte, um alles, was hier beschrieben ist, auszuprobieren. Dafür kann man aber verschiedene Konzepte nebeneinander sehen, die sich sonst nur in getrennten Sprachen jeweils für sich alleine studieren lassen. Darüber hinaus sieht man Konzepte im Kontext einer Gesamtsprache, die sonst nur in losgelösten wissenschaftlichen Papieren zu betrachten sind.

Aber natürlich wäre es sehr unbefriedigend, wenn es sich bei den von uns diskutierten Themen nur um akademische Studien handeln würde. Vieles von dem, was hier an Konzepten vorgestellt wird, ist in dieser Form Gegenstand von experimentellen Implementierungen unserer Arbeitsgruppe an der Technischen Universität Berlin. Und es ist nicht ausgeschlossen, dass daraus eines Tages auch eine reale Sprache OPAL-2 wird, mit einem Compiler, den man aus dem Internet herunterladen kann ...

Ein solches Unterfangen wird nicht alleine von zwei Menschen geleistet. Es tragen im Laufe der Jahre viele dazu bei. An erster Stelle sind hier die ehemaligen und aktuellen Mitarbeiter unserer Arbeitsgruppe an der TU Berlin zu nennen, insbesondere (in alphabetischer Reihenfolge) Michael Cebulla, Klaus Didrich, Thomas Frauenstein, Wolfgang Grieskamp, Markus Lepper, Christian Maeder, Thomas Nitsche, Wolfram Schulte, Mario Südholt, Baltasar Trancon-y-Wideman, Stephan Weber und Jacob Wieland. Sie haben im Rahmen von Projekten, Doktorarbeiten und Lehrveranstaltungen viel zur Gestaltung der hier diskutierten Konzepte beigetragen. Unserer besonderer Dank gilt André Metzner für viele fruchtbare Diskussionen, sowie Dirk Reckmann und Martin Grabmüller, die Teile des Buches kritisch gelesen und durch vielfältige Kommentare verbessert haben. Vor allem aber schulden wir Stephan Frank großen Dank, der sowohl inhaltlich als auch technisch einen enormen Beitrag geleistet hat.

Wir hatten aber auch das Vergnügen, mit vielen Kollegen innerhalb und außerhalb der TU Berlin zahlreiche stimulierende Diskussionen zu führen, aus denen wichtige Anregungen für dieses Buch hervorgingen. Besonders hervorzuheben sind hier Bernd Mahr von der TU Berlin sowie Doug Smith und Dusko Pavlovic vom Kestrel Institute. Auch die Diskussionen mit den Kollegen der IFIP Working Groups 2.1 und 1.3 haben uns viele Einsichten gebracht.

Die Mitarbeiter des Springer-Verlags haben durch ihre kompetente Unterstützung viel zu der jetzigen Gestalt des Buches beigetragen.

Berlin, im März 2006 *Peter Pepper*
Petra Hofstedt

Inhaltsverzeichnis

Hinweis: Eine Errata-Liste und weitere Hinweise zu diesem Buch sind über die Web-Adresse `http://www.uebb.cs.tu-berlin.de/books/fp` zu erreichen.

0

Das Strittigste vorab: Notationen

Über Geschmack lässt sich nicht streiten.

(Sprichwort)

Natürlich lässt sich – entgegen anderslautenden Sprichwörtern – über Geschmack ganz trefflich streiten. Einer Anekdote zufolge sollen sich einst international hochreputierte Professoren wütend über die Frage gezankt haben, ob man eine Integervariable mittels *int x* oder x: *int* einführen sollte. Die erste Variante orientiert sich an der umgangssprachlichen Formulierung „die Integervariable x“ und hat von ALGOL aus ihren Weg z. B. in C, C++ und von da aus weiter in JAVA gefunden. Die zweite Variante orientiert sich an der mathematischen Notation $x \in Int$ und drang von PASCAL aus z. B. in MODULA und ADA vor.

Die Erkenntnis, dass ein solcher Streit wohl müßig ist, hat zu einer Gegenbewegung geführt, die Notation für völlig belanglos erklärte und ausschließlich über Konzepte sprach. Doch beweist die Zahl der gescheiterten Sprachen, die exzellente Konzepte mit völlig unlesbarer Syntax verbanden, dass Notation eben doch nicht ganz vernachlässigbar ist.

In der Quintessenz hat sich die Erkenntnis durchgesetzt, dass eine Sprache zwar primär an ihren Konzepten zu messen ist, dass sie diese aber in brauchbaren Notationen vermitteln muss.

Ein Buch, das sich modernes Sprachdesign zum Thema gesetzt hat, darf deshalb das Thema Notationen nicht ignorieren. Und modern heißt heute auch, dass man die Standards übernehmen muss, die von graphischen Bildschirm- und Drucksystemen sowie von Setzwerkzeugen wie TEX und Postscript vorgegeben wurden. Das führt zu einer Sicht auf Notationen, die von gängigen Mustern bei Programmiersprachen radikal abweicht.

0.1 Jenseits von ASCII

Die Designer von Programmiersprachen scheinen eine merkwürdige Affinität zu den guten alten ASCII-Terminals zu haben. Anders ist es kaum zu erklären, dass die größte notationelle Revolution der letzten dreißig Jahre das Zulassen von UNICODE-Zeichen in JAVA war. Dabei gibt es spätestens seit den Zeiten von TEX und MATHEMATICA keinen Grund mehr, an dieser Beschränkung festzuhalten. Und wer jemals einen Kollegen in China beobachtet hat, wie er seitenweise chinesischen Text aus einem ASCII-Terminal zaubert, muss zugeben, dass die Konformität zwischen Input und lesbarem Output auch nur ein vorgeschobenes Argument ist.

Mathematische Notation

Vor diesem Hintergrund machen sich einige Informatiker wie z. B. Guy Steele von der Firma SUN inzwischen dafür stark, die Schreibweisen von Programmiersprachen radikal zu modernisieren. Es gibt schließlich keinen Grund, weshalb Programme so viel altmodischer aussehen müssen als Mathematik. Tabelle 0.1 zeigt, dass derselbe Programmtext in ganz verschiedenen Notationen dargestellt werden kann.

ASCII	UNICODE	TEX
`rho0 = r DOT r`	$\rho_0 = \mathtt{r} \cdot \mathtt{r}$	$\rho_0 = r \cdot r$
`v__norm = v / norm v`	`v__norm = v / ||v||`	$v_{norm} = \frac{v}{\|v\|}$
`SUM[k=1:n] a[k] x ^ k`	`∑[k=1:n] a[k] x ^ k`	$\sum_{k=1}^{n} a_k x^k$

Tab. 0.1: Verschiedene Schreibweisen eines Programmfragments (nach G. Steele)

Wir werden uns in diesem Buch weitestgehend an die dritte – also die fortschrittlichste – dieser Varianten halten.

Fonts

Neben mathematischen Zeichen bieten moderne Textsysteme auch die Möglichkeit, mit unterschiedlichen Fonts zu arbeiten. Allerdings besteht dabei die Gefahr, zu viel des Guten zu tun und ein extrem unruhiges Satzbild zu erzeugen. Wir suchen hier einen Kompromiss und verwenden die Fonts und Konventionen aus Tabelle 0.2.

Namen von Werten, Funktionen etc. schreiben wir (meist) klein und – mathematischer Tradition folgend – kursiv. Wie auch bei allen anderen Symbolen erlauben wir Indizierung ebenso wie Annotation mit Strichen. Typen und Strukturen werden ebenfalls kursiv geschrieben, allerdings beginnend mit einem Großbuchstaben. Für Typklassen verwenden wir einen speziellen Font.

Schlüsselwörter	FUN, DEF, TYPE, ...
Konstanten, Parameter, Funktionen	x, x_i, x', *sin*, *sqrt*, ...
Typen und Strukturen	*Int*, *Real*, *List*(*Int*), *Numbers*, ...
Typklassen	*Ord*, *Interval*, *Monoid*, *Printable*, ...

Tab. 0.2: Die Verwendung von Fonts

0.2 Jenseits von Infix: Mixfix

Die meisten Programmiersprachen kennen Infix-, Postfix und Präfix-Operatoren, so dass es problemlos möglich ist, einen Ausdruck wie „$a * -(b + c \,/\, d)$“ zu schreiben. Dabei werden sogar Präzedenzen wie die zwischen „ $*$ “ und „ $+$ “ richtig erkannt.

Allerdings gibt es diesen Komfort bei den meisten Sprachen nur für einige ausgewählte, vorgegebene Operatoren. Der Programmierer kann nur in den seltensten Fällen seine eigenen Operatoren einführen. Diese Restriktion gibt es im Wesentlichen, weil Programmierer-definierte Operatoren die Compiler komplizierter machen. Aber bei den heutigen leistungsfähigen Rechnern ist das schon lange kein Argument mehr, so dass Techniken zur Analyse der entsprechenden Programme verfügbar sind [98, 145].

In einer modernen Sprache muss der Programmierer die Chance haben, sehr flexibel eigene Notationen einzuführen.[1] Deshalb sehen einige Sprachen (z. B. ISABELLE [110, 103], MAUDE [35] oder SDF [72]) entsprechende Möglichkeiten vor, die wir hier übernehmen. Wir erlauben also nicht nur die klassischen Prä-, Post- und Infix-Operatoren, sondern ganz allgemeine ***Mixfix***-Operatoren. Die folgenden Beispiele illustrieren die Notation, wobei der *Underscore* „ _ “ die Positionen der Argumente repräsentiert.

```
FUN ¬_: Bool → Bool                          -- Präfix (Negation)
FUN _!: Nat → Nat                            -- Postfix (Fakultät)
FUN _ + _: Real × Real → Real                -- Infix (Addition)
FUN _ with _ at _: Vector × Real × Int → Vector   -- Mixfix (Update)
FUN _ ≤ _ ≤ _: Real × Real × Real → Bool     -- Mixfix (Vergleich)
```

Der Gewinn an Eleganz und Lesbarkeit zeigt sich z. B. bei der Definition des letzten der obigen Operatoren:

```
DEF a ≤ b ≤ c = (a ≤ b) ∧ (b ≤ c)
```

Wenn wir eine solche Mixfix-Operation insgesamt ansprechen – z. B. wenn sie als Argument einer Funktion höherer Ordnung auftritt – dann dürfen wir die äußersten „ _ “ weglassen, z. B.:

[1] Allerdings besteht dann das Risiko, dass Programme durch eine zu große Fülle von neuen Operatoren völlig unleserlich gemacht werden. Solchen Missbrauch zu unterbinden kann aber nicht die Aufgabe des Compilers sein. Schließlich ist ein Compiler keine Instanz zur Bewahrung des guten Geschmacks.

DEF $sum(l) = reduce(\_ + \_)(l)$ $\widehat{=}$ DEF $sum(l) = reduce(+)(l)$
DEF $ascending(l) = sort(\_ < \_)(l)$ $\widehat{=}$ DEF $ascending(l) = sort(<)(l)$
IMPORT $\_\,with\,\_\,at\,\_$ $\widehat{=}$ IMPORT $with\,\_\,at$

Man kann sogar noch weiter gehen und Operatoren einführen, die *gar keine* sichtbaren Partikel mehr enthalten. Ein sehr nützliches Beispiel ist

FUN $\_\,\_$: $Int \times Int \to Int$ $--$ *Multiplikation (unsichtbarer Operator)*

Damit lassen sich die üblichen mathematischen Notationen wie $2\ x + 3\ y$ oder auch $2\ a\ b$ in Programmen schreiben. (Man beachte die Leerzeichen!) Solche unsichtbaren Operatoren eignen sich besonders gut für Casting, etwa in der Form

FUN $\_$: $Real \to Complex$ $--$ *Casting (als unsichtbarer Operator)*

Aber Vorsicht! Dabei bewegt man sich auf sehr dünnem Eis. Eine polymorphe Funktion der Bauart FUN $\_ : \alpha \to \beta$ wäre überall und beliebig oft anwendbar und würde damit den Compiler in eine unendliche Analyse treiben.

0.3 Overloading extrem

Eine ASCII-Tastatur hat nur sehr wenige Zeichen; deshalb erlauben viele Programmiersprachen, das gleiche Symbol für unterschiedliche Operationen zu verwenden. Ein klassisches Beispiel ist das „+“-Zeichen: Es dient nicht nur dazu, alle Additionsoperationen auf Typen wie *Int*, *Real* etc. zu beschreiben, sondern wird auch (z. B. in JAVA) für die String-Konkatenation benutzt. Diese Mehrfachverwendung – ***Overloading*** oder ***Überlagerung*** (und manchmal auch *Ad-hoc-Polymorphie*) genannt – wird allerdings in den meisten Fällen nur auf eine kleine Auswahl vorgegebener Operatoren wie „+“, „*“ etc. beschränkt.

Unserem generellen Ansatz entsprechend werden wir auch hier modernem Sprachdesign folgen, das dem Programmierer die uneingeschränkte Möglichkeit zum Overloading erlaubt. Man kann also die folgenden Definitionen alle im gleichen Kontext haben:

FUN $\_ * \_$: $Int \times Int \to Int$ $--$ *Integer-Multiplikation*
FUN $\_ * \_$: $Real \times Real \to Real$ $--$ *Real-Multiplikation*
FUN $\_ * \_$: $(\alpha \to \alpha) \times Int \to (\alpha \to \alpha)$ $--$ *iterierte Funktion* f^n
FUN $\_*$: $(\alpha \to \beta) \to Seq\ \alpha \to Seq\ \beta$ $--$ *Apply-to-All (Map)*

Für den Compiler bedeutet eine solche Überlagerung natürlich eine gewaltige Herausforderung, weil der Programmierer leicht Mehrdeutigkeiten erzeugen kann. Aber erfahrungsgemäß reichen im Bedarfsfall ein oder zwei Typannotationen aus, um wieder alles eindeutig zu machen.

Wir werden die Idee des Overloadings sehr extrem ausbauen, indem wir sogar ein zentrales Sprachkonstrukt überlagern: Der Doppelpunkt „:“ kann bei uns in drei Rollen auftreten:

$\ldots$ LET x: $Nat = 5$: Nat IN $\ldots$ $--$ *Deklaration mit Typfestlegung*
$\ldots sqrt(x{:}\,Real)\ldots$ $--$ *Casting*
$\ldots$ IF x: Nat THEN $\ldots$ $--$ *Typtest*

Diese Idee ist sehr experimentell und es muss erst noch eruiert werden, ob praktische Compiler dazu tatsächlich in der Lage sind (weil die Typen selbst ein zentrales Mittel der *Overload-Resolution* sind). Aber in unserem konzeptuell orientierten Buch schätzen wir die Leistungsfähigkeit von Compilern grundsätzlich optimistisch ein.

0.4 Layout mit Bedeutung

In der Frühzeit der Programmierung (z. B. in frühen FORTRAN-Versionen) hatte die Position der Symbole essenzielle Bedeutung („alles ab Spalte 72 ist Kommentar“). Doch schon bald setzte sich das Prinzip durch, dass das Layout keinerlei Bedeutung haben sollte. Aber diese rigorose Sicht wurde in jüngerer Zeit wieder aufgegeben. In Sprachen wie OCCAM und später auch in HASKELL wurde die so genannte *Offside-Regel* eingeführt. Damit ist es möglich, Trennzeichen wegzulassen, weil sie durch das Layout „evident“ sind.

Ein typisches Beispiel ist das WHERE-Konstrukt. Man betrachte das folgende artifizielle Programmfragment; es ist zwar hässlich, aber aufgrund der Einrückungen analysierbar.

$$\begin{array}{lll} f(g\ a, y)\ \text{WHERE}\ y & = & h\ a \\ & & \quad b + c \\ \quad g\ x & = & h\ x\ a \end{array}$$

Würde man das Ganze in eine Zeile schreiben, müsste man Trennzeichen verwenden:

$f(g\ a, y)$ WHERE $y = h\ a\ b + c,\ g\ x = h\ x\ a$

Ohne das Komma wäre völlig unklar, welche Rolle z. B. g spielt; es könnte auch ein weiteres Argument von h sein.

Wir werden davon z. B. bei Summentypen Gebrauch machen. Man schreibt sie normalerweise mit dem Summenoperator „ | “.

TYPE *Shape* = *Point* | *Line* | *Circle*

Aber bei entsprechendenem Layout können wir auf diesen Operator auch verzichten:

TYPE *Shape* = *Point*
Line
Circle

Da die Programmtexte in diesem Buch primär für menschliche Leser und nicht für maschinelle Compiler geschrieben sind, werden wir das Layout sehr stark zur Unterstützung des Verständnisses einsetzen.

0.5 Bindung von Variablen

In modernen funktionalen Sprachen spielen *musterbasierte Definitionen* eine große Rolle. Dabei werden Variablen eingeführt, die über das Muster gebunden werden (vgl. Kapitel 1 und Kapitel 8). Allerdings ist bei Notationen wie

DEF $length(ft \cdot{:}\, rt) = \ldots$

oder

FUN $gen\colon (n\colon Nat \mapsto Array\,(1..n)\,Real)$

nicht ohne weiteres klar, dass z. B. *length* die zu definierende Funktion ist, während *ft* und *rt* neue, durch das Muster gebunden eingeführte Variablen sind. (Mehr dazu in Abschnitt 1.1.7 und in Abschnitt 8.2.) Analog verhält es sich mit der Funktion *gen* und der gebundenen Variablen n.

Wir gehen in den meisten Fällen davon aus, dass die Natur der jeweiligen Symbole durch den Kontext klar ist. Falls wir aber das Bedürfnis haben, die Einführung von gebundenen Variablen explizit auszuweisen, dann schreiben wir das in folgender Form:[2]

DEF $length(\widehat{ft} \cdot{:}\, \widehat{rt}) = 1 + length(rt)$

bzw.

FUN $gen\colon (\widehat{n}\colon Nat \mapsto Array\,(1..n)\,Real)$

Man beachte, dass dabei nur die Bindungsstelle explizit gekennzeichnet wird. An allen übrigen Stellen wird der Variablenname normal geschrieben.

In eher typtheoretisch orientierten Büchern spricht man beim Typ der Funktion *gen* von einem so genannten *abhängigen Typ* (engl.: *dependent type*), weil der *Typ* des Resultats vom *Wert* des Arguments abhängt. Die dazu nötige Variable wird dann oft in Anlehnung an die λ-Notation mit dem Symbol Π gebunden:

FUN $gen\colon (\Pi\, n\colon Nat \bullet Array\,(1..n)\,Real)$

Die übliche Schreibweise $A \to B$ ist dann nur eine Abkürzung für den Typterm $\Pi\, a\colon A \bullet B$, sofern die Variable a nicht im Typausdruck B vorkommt. Genauer wird das in Kapitel 8 diskutiert.

Eine geschwätzige Notation

Wenn die Variablenbindungen etwas komplexer sind, werden die obigen Kompaktnotationen schnell unleserlich. Dann beginnen eher längliche Schreibweisen besser lesbar zu sein. Betrachten wir zwei Beispiele. Das erste ist eine

[2] Dazu gibt es eine hübsche Anekdote zum Lambda-Kalkül. Es heißt, dass Church ursprünglich genau diese Notation benutzt hat (in Anlehnung an eine frühere Schreibweise von Russell). Aber der Setzer konnte das Zeichen $\widehat{\ }$ nicht über einen Buchstaben setzen, und davor sah es hässlich aus. Deshalb hat er schließlich das ähnlichste Zeichen vorangestellt, das er besaß – und das war das λ.

Operation, die wir in Kapitel 14 kennenlernen werden; mit ihr kann man ein Element an einen Array anhängen:

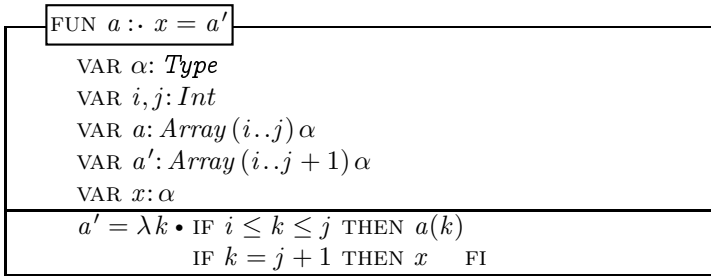
$$\begin{array}{l}\textsc{fun}\ \_ :\cdot \_ :\ [\alpha\colon \mathit{Type}] \mapsto [i\colon \mathit{Int}, j\colon \mathit{Int}] \mapsto \\ \qquad\qquad \mathit{Array}\,(i..j)\,\alpha \times \alpha \to \mathit{Array}\,(i..j+1)\,\alpha \\ \textsc{def}\ a :\cdot x = \lambda k \bullet \textsc{if}\ i \le k \le j\ \textsc{then}\ a(k) \\ \qquad\qquad\qquad\qquad \textsc{if}\ k = j+1\ \textsc{then}\ x\ \textsc{fi} \quad \textsc{where}\ (i,j) = \mathit{dom}\ a\end{array}$$

Hier kommen mehrere gebundene Variablen vor: α ist ein so genannter *polymorpher Typ*, i und j werden zur Beschreibung des *abhängigen Typs* benötigt, und a und x sind Variablen, die für die *musterbasierte Definition* gebraucht werden.

Wenn wir die Bindungen der Variablen explizit auszeichnen wollen, schreiben wir

$$\begin{array}{l}\textsc{fun}\ \_ :\cdot \_ :\ [\widehat{\alpha}\colon \mathit{Type}] \mapsto [\widehat{i}\colon \mathit{Int}, \widehat{j}\colon \mathit{Int}] \mapsto \\ \qquad\qquad \mathit{Array}\,(i..j)\,\alpha \times \alpha \to \mathit{Array}\,(i..j+1)\,\alpha \\ \textsc{def}\ \widehat{a} :\cdot \widehat{x} = \lambda \widehat{k} \bullet \textsc{if}\ i \le k \le j\ \textsc{then}\ a(k) \\ \qquad\qquad\qquad\qquad \textsc{if}\ k = j+1\ \textsc{then}\ x\ \textsc{fi} \quad \textsc{where}\ (i,j) = \mathit{dom}\ a\end{array}$$

Wenn man sich eine Sprache wie Z ansieht, dann wird dort für solche Situationen eine zweidimensionale Notation mit kleinen graphischen Verzierungen benutzt. Übertragen auf unsere Situation könnte das dann etwa so aussehen:



1

Grundlagen der Funktionalen Programmierung

Durch das Einfache geht der Eingang zur Wahrheit.
Lichtenberg (Vermischte Schriften)

Wir wollen als Erstes einen ganz knappen Überblick über die fundamentalen Konzepte der Funktionalen Programmierung geben, nämlich Funktionen und Typen. Dazu gehört insbesondere ein kurzer Abriss zum Konzept der Funktionen höherer Ordnung, die ganz wesentlich zur Eleganz und zur Produktivität dieses Paradigmas beitragen. Dieses Kapitel dient vor allem der Auffrischung von elementaren Konzepten der Funktionalen Programmierung, wie sie in zahlreichen Lehrbüchern über ML, HASKELL, OPAL und andere Sprachen nachzulesen sind (z. B. [22, 50, 102, 138, 20, 81, 111, 32]).

1.1 Funktionen

Die Funktionale Programmierung basiert – nicht überraschend – auf dem Funktionsbegriff der Mathematik. Eine ***Funktion*** bildet Eingabewerte, d. h. Elemente aus dem *Definitionsbereich*, eindeutig auf Ausgabewerte, d. h. Elemente aus dem *Wertebereich*, ab.

In erster Näherung liefert das eine ganz simple Sicht auf funktionale Programme: Ein solches Programm besteht im Wesentlichen aus einer Sammlung von Funktionsdefinitionen, und ein Aufruf des Programms erfolgt durch die Applikation einer dieser Funktionen auf Eingabewerte.

Wir werden im Laufe dieses Buches natürlich noch sehen, dass es eine Fülle von Erweiterungen, Variationen und Spezialitäten gibt, aber in letzter Konsequenz wird sich zeigen, dass alles immer wieder auf diese elementare Sichtweise hinausläuft. Deshalb wollen wir im Folgenden diese elementaren Bausteine etwas genauer betrachten.

1.1.1 Funktionsdefinition

Ein funktionales ***Programm*** ist letztlich eine Menge von Funktionsdefinitionen. Für diese Definitionen gibt es eine Reihe von Schreibweisen, die wir im Folgenden kurz vergleichen wollen. Wir beginnen mit der heute beliebtesten Notation, die vor allem durch HASKELL populär gemacht wurde.

Definition (Funktionsdefinition)

Eine ***Funktionsdefinition*** erfolgt in der Form

$$\text{DEF } f\ x_1 \ldots x_n = e.$$

Hier wird eine Funktion mit Namen f eingeführt; die Namen $x_1, \ldots, x_n$ sind ***Parameter*** und der Ausdruck e wird als ***Rumpf*** bezeichnet. Ein oder mehrere der Parameter x_i können auch Tupel $(x_{i_1}, \ldots, x_{i_n})$ sein.

Als einfaches Beispiel einer solchen Funktionsdefinition betrachten wir die Berechnung von Dreiecksflächen, und zwar bei einem gleichseitigen Dreieck mit Seitenlänge a, bei einem rechtwinkligen Dreieck mit den Katheten a und b sowie bei einem allgemeinen Dreieck mit den Seiten a, b und c.

$$\begin{array}{lll}
\text{DEF } \mathit{area}\ a & = \frac{a^2}{4}\sqrt{3} & -- \textit{ gleichseitiges Dreieck} \\
\text{DEF } \mathit{area}\ a\ b & = \frac{a\ b}{2} & -- \textit{ rechtwinkliges Dreieck} \\
\text{DEF } \mathit{area}\ a\ b\ c & = \sqrt{s(s-a)(s-b)(s-c)} & -- \textit{ allgemeines Dreieck} \\
 & \quad \text{WHERE } s = \frac{a+b+c}{2} &
\end{array}$$

Diese Art der Darstellung ist ein Spezialfall der so genannten *musterbasierten Definition*, auf die wir in Abschnitt 1.1.7 gleich noch genauer eingehen werden. Außerdem sei schon jetzt darauf hingewiesen, dass diese Form sich auf subtile Weise von der Tupelform

$$\begin{array}{l}
\text{DEF } \mathit{area}(a,b) = \frac{a\ b}{2} \\
\text{DEF } \mathit{area}(a,b,c) = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{WHERE } s = \frac{a+b+c}{2}
\end{array}$$

unterscheidet. Mehr dazu in Abschnitt 1.1.5.

Man beachte auch, dass wir hier ganz im Sinne der syntaktischen Verabredungen des vorigen Kapitels den Rumpfausdruck nicht in altmodischer ASCII-Notation schreiben, sondern in eleganter mathematischer Notation.

Die Beispiele illustrieren auch das Prinzip des *Overloading*: Alle drei Funktionen heißen *area*; welche jeweils gemeint ist, lässt sich anhand der Parameterzahl erkennen.

Die obige Definition schließt auch die notationelle Variante der Mixfixoperatoren (s. Abschnitt 0.2) ein. Dabei ergibt sich höchstens das compilertechnische Problem, zu erkennen, was Funktionsname ist und was Parameter.

1.1.2 Ausdrücke

Das dritte der obigen *area*-Beispiele zeigt ein weiteres Sprachelement: Der Rumpfausdruck lässt sich durch *lokale Deklarationen* strukturieren, indem mittels WHERE-Klauseln abkürzende Namen für Teilausdrücke eingeführt werden. Man kann Deklarationen auch in Form von LET-IN-Klauseln voranstellen:

$$\begin{aligned}&\text{DEF } area\ a\ b\ c = \text{LET}\\ &\qquad\qquad s = \tfrac{a+b+c}{2}\\ &\qquad\text{IN}\\ &\qquad\qquad \sqrt{s(s-a)(s-b)(s-c)}\end{aligned}$$

Bei LET- und WHERE-Klauseln können mehrere Namen deklariert werden, die sich aufeinander beziehen dürfen. Die Regeln für die Reihenfolge der Deklarationen unterscheiden sich in den einzelnen Sprachen. Sie ist z. B. in OPAL beliebig, während in ML das Prinzip „Deklaration vor Verwendung“ eingehalten werden muss. OPAL verbietet zyklische Abhängigkeiten, während HASKELL sie erlaubt und ML dafür das Schlüsselwort `letrec` vorsieht.

Der Ausdruck im Rumpf kann neben der einfachen Komposition von Funktionen und Operatoren auch noch *Fallunterscheidungen* enthalten. Ein typisches Beispiel ist das Maximum zweier Zahlen:

$$\text{DEF } max\ a\ b = \text{IF } a \geq b \text{ THEN } a \text{ ELSE } b \text{ FI}$$

Durch Schachtelung solcher IF-THEN-ELSE-Ausdrücke lassen sich beliebig viele Fälle unterscheiden. Allerdings sieht z. B. OPAL eine Variante vor, die im Stil von E. W. Dijkstras *Guarded Commands* gleichberechtigte Fälle deutlicher als solche charakterisiert:

$$\begin{aligned}&\text{DEF } max\ a\ b = \text{IF } a \geq b \text{ THEN } a\\ &\qquad\qquad\qquad\text{IF } b \geq a \text{ THEN } b \text{ FI}\end{aligned}$$

Hier bleibt es dem Compiler überlassen, welchen der beiden Zweige er im Fall $a = b$ zur Berechnung auswählt. Die Eleganz dieser Variante zeigt sich deutlich, wenn wir z. B. das Maximum dreier Zahlen bestimmen wollen:

$$\begin{aligned}&\text{DEF } max\ a\ b\ c = \text{IF } a \geq b \wedge a \geq c \text{ THEN } a\\ &\qquad\qquad\qquad\qquad\text{IF } b \geq a \wedge b \geq c \text{ THEN } b\\ &\qquad\qquad\qquad\qquad\text{IF } c \geq a \wedge c \geq b \text{ THEN } c \text{ FI}\end{aligned}$$

1.1.3 λ-Notation

Funktionale Sprachen erlauben neben der oben gezeigten gleichungsartigen Funktionsdefinition, die an mathematische Gleichungen erinnert, üblicherweise auch die so genannte ***λ-Notation***. Der Ursprung dieser Notation[1] ist der λ-Kalkül [34, 17] von Church, der als theoretische Basis der funktionalen Sprachen betrachtet werden kann.

[1] Zur Entstehungsgeschichte des Zeichens „λ“ s. die Fußnote 2 auf Seite 8.

Definition (Funktionsdefinition in λ-Notation)

Eine ***Funktionsdefinition in λ-Notation*** erfolgt in der Form

DEF $f = \lambda x_1, \ldots, x_n \bullet e$

Durch das Fragment $\lambda x_1, \ldots, x_n$ werden die Variablen $x_1, \ldots, x_n$ im Ausdruck e gebunden. Den gesamten Ausdruck $\lambda x_1, \ldots, x_n \bullet e$ bezeichnet man als ***λ-Ausdruck*** oder auch als ***λ-Term***.

Die beiden Formen der Funktionsdefinition – musterbasiert und als λ-Term – sind völlig gleichwertig; sie unterscheiden sich bestenfalls als Geschmacksfrage.

Die obigen Beispiele der Dreiecksfläche (in der Tupelvariante) sehen in dieser Notation folgendermaßen aus (s. auch Abschnitt 1.1.5):

$$\begin{array}{ll}
\text{DEF } \mathit{area} = \lambda\, a \bullet \frac{a^2}{4}\sqrt{3} & -\!-\ \textit{gleichseitiges Dreieck} \\
\text{DEF } \mathit{area} = \lambda\, a, b \bullet \frac{a\ b}{2} & -\!-\ \textit{rechtwinkliges Dreieck} \\
\text{DEF } \mathit{area} = \lambda\, a, b, c \bullet \sqrt{s(s-a)(s-b)(s-c)} & -\!-\ \textit{allgemeines Dreieck} \\
\qquad \text{WHERE } s = \frac{a+b+c}{2} &
\end{array}$$

Bei der Funktionsdefinition stellt die λ-Notation nur eine alternative Schreibweise dar. Aber sie hat einen anderen wichtigen Vorteil: Mit ihr kann man Funktionen auch direkt aufschreiben und verwenden, ohne ihnen dazu einen expliziten Namen geben zu müssen. Solche *anonymen Funktionen* können z. B. dann genutzt werden, wenn man mit Funktionen als Argumenten arbeitet. Ein typisches Beispiel ist die Funktion *filter*, die wir in Abschnitt 1.2 gleich noch genauer betrachten werden:

$$\ldots \mathit{filter}\ (\lambda\, x \bullet x < 0\,)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots$$

Diese Funktion behält nur diejenigen Elemente einer Liste, die das Prädikat $(\lambda\, x \bullet x < 0\,)$ erfüllen, d. h. alle negativen Zahlen. Im Beispiel der obigen Liste $\langle 3, 5, -2, 0, 18, -12\rangle$ ist das Ergebnis also die Liste $\langle -2, -12\rangle$.

Als Kurzform verwendet z. B. die Sprache OPAL auch die ***Wildcard-Notation***, also $(\_\ \hat{}\ 2)$ als Kurzform für den λ-Term $(\lambda\, x \bullet x\ \hat{}\ 2)$. In anderen Sprachen wird das über so genannte *Sections* formuliert, d. h. in der Form $(\hat{}\ 2)$. Wir werden beide Versionen verwenden. Denn insbesondere das letztere Beispiel entsteht auch durch die Konvention von Kapitel 0, dass man den Platzhalter „_“ am Anfang und Ende eines Mixfix-Symbols weglassen darf. Die obige Anwendung der Funktion *filter* kann also in den folgenden drei gleichwertigen Formen geschrieben werden:

$$\begin{array}{l}
\ldots \mathit{filter}\ (\lambda\, x \bullet x < 0)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots \\
\ldots \mathit{filter}\ (_ < 0)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots \\
\ldots \mathit{filter}\ (< 0)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots
\end{array}$$

Man beachte, dass diese Kurzform mittels „_“ nur bei Funktionen mit einem Parameter anwendbar ist; diese treten aber besonders häufig auf.

1.1.4 Funktionalitäten: Die Typisierung von Funktionen

Typen beschreiben die Funktionalität von Funktionen (und Ausdrücken), d. h. die Art (und Anzahl) der Eingabe- und Ausgabewerte.

Definition (Typ, Funktionalität)

Der ***Typ*** einer Funktion (ihre ***Funktionalität***) wird im Allgemeinen folgendermaßen geschrieben:

FUN f: $A \to B$ −− *Funktion*
VAL c: B −− *Konstante*

- Der Definitionsbereich der Funktion f ist A und der Wertebereich ist B. Wie in der Mathematik üblich werden beide durch einen Pfeil getrennt.
- Als Grenzfall von Funktionen haben wir auch *Konstanten*. Hier ist der Definitionsbereich leer. Aus Gründen der Lesbarkeit lassen wir den Pfeil weg und geben nur den Wertebereich B an. Als Schlüsselwort schreiben wir dann manchmal auch VAL anstelle von FUN.
- Definitions- und Wertebereich brauchen nicht nur aus elementaren Typen zu bestehen, sondern können auch Typausdrücke[2] sein, die aus *Produkttypen* $(A_1 \times \ldots \times A_n)$ und *Funktionstypen* $(A \to B)$ aufgebaut sind. Im letzteren Fall sprechen wir von *Funktionen höherer Ordnung*. Für diese potenziell sehr komplexen Typausdrücke gelten folgende Klammerregeln:
  - Die Tupelbildung „$\times$“ bindet stärker als die Funktionsbildung „ $\to$ “.
  - Die Funktionsbildung „ $\to$ “ bindet nach rechts, d. h., $A \to B \to C$ ist gleichbedeutend zu $A \to (B \to C)$.
- ***Sorten*** bilden die Basis für die Typbeschreibung. Das sind die Namen für elementare Mengen (wie *Nat*, *Int*, *Real*, *Char*, *String* usw.)

Betrachten wir als Beispiel wieder die Berechnung des Flächeninhalts von Dreiecken. Wir geben zu den Definitionen die jeweiligen Funktionalitäten mit an. In der Tupelvariante ergibt sich:

FUN $area$: $Real \to Real$ −− *gleichseitiges Dreieck*
DEF $area\ a = \frac{a^2}{4}\sqrt{3}$

FUN $area$: $Real \times Real \to Real$ −− *rechtwinkliges Dreieck*
DEF $area(a, b) = \frac{a\ b}{2}$

FUN $area$: $Real \times Real \times Real \to Real$ −− *allgemeines Dreieck*
DEF $area(a, b, c) = \sqrt{s(s-a)(s-b)(s-c)}$
WHERE $s = \frac{a+b+c}{2}$

Man kann das Gleiche auch in der Form von Funktionen höherer Ordnung schreiben (s. Abschnitt 1.1.5):

[2] Eine detaillierte Diskussion von Typen und Typausdrücken werden wir in den Kapiteln 6-9 führen.

FUN *area*: $Real \to Real \to Real$ −− *rechtwinkliges Dreieck*
DEF $area\ a\ b = \frac{a\ b}{2}$

FUN *area*: $Real \to Real \to Real \to Real$ −− *allgemeines Dreieck*
DEF $area\ a\ b\ c = \sqrt{s(s-a)(s-b)(s-c)}$
WHERE $s = \frac{a+b+c}{2}$

Übrigens: Die Funktionalitäten zeigen das *Overloading* besonders deutlich.

1.1.5 Partielle Applikation und Currying

Funktionale Sprachen erlauben es, Funktionen auf weniger Argumente anzuwenden als nötig. Man nennt dies eine ***partielle Applikation***. Durch die partielle Applikation einer Funktion erhält man eine neue Funktion, die bei Anwendung auf die restlichen Argumente das gleiche Ergebnis wie die ursprüngliche vollständig applizierte Funktion liefert.

Um die partielle Applikation einer Funktion zu erlauben, müssen wir bei der Definition der Funktionalität anstelle des Produkts „ × “ den Funktionspfeil „ → “ schreiben. Diesen Übergang von der Tupelbildung zum Funktionspfeil nennt man ***Currying***, die dadurch entstehende neue Notation ***Curry-Notation***. Das haben wir in den obigen Beispielen der *area*-Funktionen gerade gesehen:

FUN *area*: $Real \times Real \to Real$ −− *Tupelversion*
DEF $area(a, b) = \frac{a\ b}{2}$

FUN *area*: $Real \to Real \to Real$ −− *Curry-Version*
DEF $area\ a\ b = \frac{a\ b}{2}$

Generell wird aus dem n-stelligen Tupeltyp $T_1 \times T_2 \times \ldots \times T_n \to T$ die Curry-Form $T_1 \to (\, T_2 \to \ldots \to (T_n \to T) \ldots)$. Mathematisch sind beide Typen isomorph.

Wenn wir die Curry-Version von *area* partiell applizieren, z. B. *area* 1.0, erhalten wir eine Funktion, die bei Eingabe der zweiten Kathete den Flächeninhalt des rechtwinkligen Dreiecks berechnet. Das kann sogar in eine entsprechende Definition gefasst werden:

FUN *partialArea*: $Real \to Real$ −− *braucht nur noch die zweite Kathete*
DEF $partialArea = area\ 1.0$

Die Funktion *partialArea* erlaubt also die Flächenberechnung für rechtwinklige Dreiecke, bei denen eine Kathete die vorgegebene Länge 1.0 hat. Die Definition könnte übrigens auch in der Form

DEF $partialArea\ b = area\ 1.0\ b$

oder in λ-Form

DEF $partialArea = \lambda\, b \bullet area\ 1.0\ b$

geschrieben werden. (Im λ-Kalkül spricht man hier von η-Reduktion.)

1.1.6 Typen

Das Thema *Typisierung* ist ein besonders aktives Forschungsfeld bei den modernen Programmiersprachen. Deshalb diskutieren wir es intensiv in den Kapiteln 6 bis 9. Für diesen einführenden Abschnitt beschränken wir uns auf ein eher intuitives Verständnis der elementarsten Konstruktionen.

Produkttypen (Tupeltypen)

Mit Produkttypen (auch *Tupeltypen* genannt) kann man logisch zusammengehörende Daten gleichen oder verschiedenen Typs in einem neuen Datentyp zusammenfassen.

Beispiel: Ein Punkt im zweidimensionalen Raum ist durch seine Koordinaten x und y charakterisiert. Wir definieren den Tupeltyp *Point*:

TYPE $Point = Real \times Real$ $--$ *ohne Konstruktoren und Selektoren*

Will man auf die einzelnen Elemente des Tupeltyps direkt zugreifen, kann man bei der Typdefinition auch Selektoren angeben. (Die Verwendung des Gleichheitszeichens anstelle des Doppelpunkts, der hier in den meisten Sprachen steht, wird in Kapitel 6 eingehend erläutert werden.)

TYPE $Point = (x = Real, y = Real)$ $--$ *... mit Selektoren ...*

Dadurch erhalten wir die *Selektionsfunktionen* x und y, die wir auf einen Punkt p anwenden können. Mit $x(p)$ greifen wir dann beispielsweise auf die erste Komponente des Punktes p zu. (Wie wir gleich noch in Abschnitt 1.2 sehen werden, können wir das auch als $p.x$ schreiben – was eher an Notationen aus Sprachen wie JAVA erinnert.)

Schließlich können wir auch noch eine *Konstruktorfunktion*, hier *point*, angeben; in diesem Fall sprechen wir von ***Konstruktortypen***:

TYPE $Point = point(x = Real, y = Real)$ $--$ *... sowie mit Konstruktor*

Das ist besonders im Zusammenhang mit dem so genannten *Patternmatching* nützlich, wie wir gleich noch sehen werden.

Summentypen

Neben Produkttypen braucht man auch ***Summentypen***, um inhaltlich verwandte Elemente, die aber strukturell unterschiedlich aufgebaut sein können, zusammenzufassen.

Betrachten wir neben Punkten weitere geometrische Elemente, die wir als entsprechende Typen definiert haben, dann können wir diese in einem Typ *Shape* zusammenfassen:

TYPE $Shape = Point \mid Line \mid Circle \mid Triangle \mid Rectangle$

Dann können wir immer da, wo ein Wert vom Typ *Shape* erwartet wird, Werte der Typen *Point*, *Line* etc. benutzen (Näheres in Abschnitt 6.5).

Ein besonders häufiger Spezialfall ist die Bildung von Summentypen, deren Varianten direkt als Konstruktortypen angegeben sind. Zum Beispiel können wir für einen Punkt zwei Darstellungen vorsehen:

TYPE *Point* = *koord*(*x* = *Real*, *y* = *Real*)
| *polar*(*dist* = *Real*, *angle* = *Real*)

Die Konstruktoren können sogar mittels Overloading gleich benannt werden, wie das folgende Beispiel illustriert.

TYPE *Line* = *line*(p_1 = *Point*, p_2 = *Point*)
| *line*(*p* = *Point*, *angle* = *Real*, *length* = *Real*)

Rekursive Typen

Häufig werden Produkt- und Summentypen kombiniert, insbesondere dann, wenn die Definition der Typen ***rekursiv*** ist.

Beispiel: Eine Sequenz reeller Zahlen ist entweder leer, hier dargestellt durch $\Diamond$, oder sie besteht aus einer reellen Zahl, die mit einer weiteren Sequenz verkettet ist:

TYPE *RealSeq* = { $\Diamond$ }
| *prepend*(*ft* = *Real*, *rt* = *RealSeq*)

Diese Definition kombiniert einen Summentyp mit einem rekursiven Produkttyp. Den Konstruktor schreiben wir übrigens meistens als Infixoperator:

TYPE *RealSeq* = { $\Diamond$ }
| _ ·: _(*ft* = *Real*, *rt* = *RealSeq*)

1.1.7 Musterbasierte Funktionsdefinition

Basierend auf der Definition von Datentypen mit Konstruktoren kann man in einfacher Weise so genannte ***musterbasierte Funktionsdefinitionen*** (engl.: *pattern-based definitions*) aufschreiben. Dies ist im Allgemeinen kürzer und eleganter als eine Definition, die auf einer Folge von IF-Abfragen basiert.

Beispiel: Die Funktion *length* berechnet die Länge einer Sequenz, indem sie diese durchläuft und die Elemente zählt.

FUN *length*: *Seq* → *Nat*
DEF *length* $\Diamond$ = 0
DEF *length*(*prepend*(*first*, *rest*)) = 1 + *length*(*rest*)

Bei einer musterbasierten Funktionsdefinition DEF $f(c(x_1, \ldots, x_n), \ldots) = e$ sind die Argumente der linken Seite Konstruktorterme $c(x_1, \ldots, x_n)$. Man nennt sie auch *Muster* oder *Pattern*. Für eine Funktion gibt man dabei (in der Regel für ein Argument) so viele DEF-Deklarationen an wie der zugehörige Summentyp Varianten aufweist. Die Variablen $x_1, \ldots, x_n$ können dann im Rumpf so benutzt werden wie bisher die Variablen der linken Seite bzw. wie die durch ein λ-Konstrukt gebundenen Variablen.

Bei der Auswertung wird der Ausdruck dann mit jedem der Pattern der entsprechenden Funktion verglichen und die Regel mit passendem Muster ausgewählt. Dieses Vorgehen nennt man ***Patternmatching***.

Das Konzept der musterbasierten Definitionen überträgt sich in analoger Weise auch auf Infixoperatoren:

DEF $length\ \Diamond = 0$
DEF $length(first \cdot\!: rest) = 1 + length(rest)$

Wenn man das Prinzip des *best-fit* Patternmatching verwendet, kann man auch folgende Art von Definitionen schreiben:

FUN fac: $Nat \to Nat$
DEF $fac\ 0 = 1$
DEF $fac\ n = n \cdot fac(n-1)$

Hier würde das zweite Pattern – das ja nur eine Variable n ist – im Prinzip immer passen. Aber das Pattern 0 – ein konstanter Konstruktor – ist spezifischer und hat daher Vorrang. (In diesem Beispiel würde auch das so genannte *first-fit* Patternmatching funktionieren, das aber den generellen Nachteil hat, dass die Aufschreibungsreihenfolge der Definitionen relevant wird.)

1.1.8 Erweitertes Patternmatching

Die Eleganz des Programmierens lässt sich noch steigern, wenn man das Prinzip des Patternmatchings so verallgemeinert, dass nicht nur Konstruktorfunktionen zugelassen werden.

Beispiel: Die folgende – ziemlich artifizielle – Funktion wurde von Dijkstra benutzt, um Beweistechniken zu illustrieren.

FUN $fusc$: $Nat \to Nat$
DEF $fusc\ 0 = 1$
DEF $fusc\ 1 = 1$
DEF $fusc(2 \cdot n) = fusc\ n$
DEF $fusc(2 \cdot n + 1) = fusc(n) + fusc(n+1)$

Das ist im Grunde nicht korrekt, da „1“, „ $+$ “ und „ $\cdot$ “ keine Konstruktoren sind. Deshalb müssen wir in die Sprache Mechanismen aufnehmen, die ein solches erweitertes Patternmatching ermöglichen. Dazu verwenden wir das spezielle Schlüsselwort MATCHES.

Im obigen Beispiel müssen für einen Aufruf $fusc(x)$ vom Compiler die vier Fälle erkennbar sein: (x MATCHES 0), (x MATCHES 1), (x MATCHES $2 \cdot n$) und (x MATCHES $2 \cdot n + 1$). In den letzten beiden Fällen muss dabei noch die freie Variable n an eine entsprechende Zahl gebunden werden. Die technischen Details lassen wir hier offen, da sie eher in Bereiche wie Compilertechnik oder Constraint-Solving führen. Wir beschränken uns auf eine Skizze der notwendigen Mechanismen.

- Man braucht ein *Prädikat*, das den Erfolg des Matchings anzeigt. Im obigen Beispiel ist das

  x MATCHES $2 \cdot n \iff even\ x$
  x MATCHES $2 \cdot n + 1 \iff odd\ x$

- Außerdem braucht man Funktionen, die den freien Variablen im Muster ihre Instanzwerte zuordnen:

  x MATCHES $2 \cdot n \implies n = \frac{x}{2}$
  x MATCHES $2 \cdot n + 1 \implies n = \frac{x-1}{2}$

Die Prädikate werden vom Compiler benutzt, um die musterbasierte Definition in eine IF-Kaskade umzuwandeln, und die Instanziierungsfunktionen führen dann in entsprechenden LET-Anweisungen die Patternvariablen ein.

Man beachte: Was wir hier für beliebige Muster sehr länglich aufschreiben müssen, wird vom Mechanismus der Summen- und Produkttypen automatisch geleistet, da der Compiler hier die entsprechenden Test- und Selektorfunktionen intern generiert.

1.1.9 Polymorphie

Kann man eine Funktion für Elemente unterschiedlichen Typs in gleicher Weise und unabhängig von ihrem Typ definieren, so spricht man von ***(parametrischer) Polymorphie***. Wir werden dieses Konzept in Kapitel 5 und Kapitel 8 ausführlich diskutieren. Für unsere einführenden Beispiele brauchen wir aber zumindest ein erstes intuitives Verständnis der Grundidee.

Die einfachste polymorphe Funktion ist die Identitätsfunktion:

FUN $id\colon \alpha \to \alpha$
DEF $id\ x = x$

Definition (polymorphe Funktion, polymorpher Datentyp, Typvariable)

Ein Datentyp ist ***polymorph***, wenn seine Definition von einem Typparameter abhängt. Der Typparameter wird auch als ***Typvariable*** bezeichnet, für die wir im Folgenden meist griechische Buchstaben $\alpha, \beta, \gamma, \ldots$ verwenden.

Eine Funktion heißt ***polymorph***, wenn ihre Funktionalität einen polymorphen Typ enthält.

Ein klassisches Beispiel sind Sequenzen über Elementen eines beliebigen Typs. Sie werden als polymorpher Datentyp definiert:

TYPE $Seq\ \alpha\ =\ \{\ \Diamond\ \}$
$\quad\quad\quad\quad\quad\quad\ \mid\ \ \_ \cdot\colon \_(ft = \alpha,\ rt = Seq\ \alpha)$

Auf solchen Sequenzen kann man polymorphe Funktionen definieren, die völlig unabhängig vom Typ ihrer Elemente sind. Ein typisches Beispiel ist die früher schon erwähnte Funktion *length*:

FUN $length$: $Seq\ \alpha \to Nat$
DEF $length(\Diamond) = 0$
DEF $length(first \cdot: rest) = 1 + length(rest)$

In Abschnitt 1.2 werden uns typische polymorphe Funktionen auf Sequenzen als Funktionen höherer Ordnung begegnen.

Anmerkung: Wie schon in Abschnitt 0.5 angesprochen, kann die Erkennung der Typvariablen unter Umständen Probleme machen. Diese Probleme werden wir in Kapitel 8 noch ausführlich diskutieren. Vorläufig umgehen wir das Problem, indem wir Typvariablen als griechische Buchstaben schreiben.

1.1.10 Eigenschaften (Propertys)

Bisher haben wir bei Funktionen die beiden klassischen Sprachkonstrukte skizziert: ihre *Definition* und ihre *Typisierung*. Und die meisten Sprachen beschränken sich auch auf diese beiden Features. Aber in vielen Fällen bräuchte man noch eine weitere Möglichkeit: Man möchte ausdrücken, dass Funktionen bestimmte Eigenschaften haben. Wir führen dazu das Sprachmittel der ***Propertys*** ein.

Beispiel: Die Sinus- und die Kosinusfunktion werden numerisch über geeignete Reihendarstellungen berechnet. Aber sie haben auch diverse mathematische Beziehungen zueinander:

DEF $sin(x) = \ldots$ «*Reihenentwicklung*» $\ldots$
DEF $cos(x) = \ldots$ «*Reihenentwicklung*» $\ldots$
PROP $cos(x) = sin(x + \frac{\pi}{2})$

Während diese Property für einen Compiler nicht viel nützen wird, gibt es zahlreiche Fälle, in denen das anders ist. Wichtige Eigenschaften von Funktionen sind z. B. die Assoziativität oder Kommutativität, die Existenz einer inversen Funktion oder eines neutralen Elements, die Idempotenz usw.

Manchmal ist es auch nützlich, die Propertys zu benennen. Dann kann man sich z. B. in Programmbeweisen oder -entwicklungen explizit auf sie beziehen:

PROP $cosToSin$ IS $cos(x) = sin(x + \frac{\pi}{2})$

Übrigens: Die Typisierung einer Funktion ist natürlich nichts anderes als eine spezielle Property. Mit anderen Worten,

FUN sin: $Real \to Real$

ist eigentlich nur eine syntaktische Variante folgender Property:

PROP sin: $Real \to Real$

Wir werden von den Propertys sehr starken Gebrauch machen, wenn wir in Kapitel 9 mit Spezifikationen und Typklassen arbeiten. In Kapitel 16 werden wir sie zur Entwicklung von Algorithmen heranziehen.

1.1.11 Sequenzen (Listen, Folgen)

Programmierparadigmen haben oft bestimmte Datenstrukturen, auf die sie besonders zugeschnitten sind. Bei imperativen Sprachen wie ALGOL, PASCAL, C oder JAVA – und ganz besonders bei FORTRAN – ist das der Array. Bei funktionalen Sprachen ist es die ***Liste***, oft auch ***Sequenz*** oder ***Folge*** genannt. Weil auch wir diese Struktur oft in Beispielen verwenden, geben wir hier kurz ihre wichtigsten Aspekte an.[3]

```
TYPE Empty = { ◊ }
TYPE Seq α  =  Empty
             |  _ ·: _(ft = α, rt = Seq α)
```

Sequenzen sind üblicherweise polymorph definiert mit den Konstruktoren „$\Diamond$“ (*empty*) und „ $\cdot\!:$ “ (*prepend*). Den Hilfstyp *Empty* führen wir der besseren Lesbarkeit halber ein.

Die Längenberechnung haben wir weiter oben schon als Beispiel programmiert. Interessant sind aber auch die Operationen „am falschen Ende“, also *front* und *last*, sowie „ $:\!\cdot$ “ (*append*).

```
FUN front: Seq α → Seq α              −− Vorderteil der Sequenz
FUN last: Seq α → α                   −− letztes Element
FUN _ :· _: Seq α × α → Seq α         −− append
DEF front(x ·: ◊) = ◊
DEF front(x ·: rest) = x ·: front(rest)
DEF last(x ·: ◊) = x
DEF last(x ·: rest) = last(rest)
DEF ◊ :· x = x ·: ◊
DEF (a ·: rest) :· x = a ·: (rest :· x)
```

Man beachte, dass diese Definitionen wesentlich Gebrauch vom best-fit Patternmatching machen. Außerdem sind *front* und *last* für die leere Sequenz natürlich nicht definiert.

Weitere nützliche Funktionen sind die Konkatenation sowie eine schöne Mixfixnotation für die einelementige Sequenz:

```
FUN _ ++ _: Seq α × Seq α → Seq α     −− Konkatenation
FUN ⟨_⟩: α → Seq α                    −− einelementige Sequenz
DEF ◊ ++ s = s
DEF (a ·: rest) ++ s = a ·: (rest ++ s)
DEF ⟨x⟩ = x ·: ◊
```

Wir werden uns oft die Freiheit nehmen, nicht nur die einelementige Sequenz, sondern auch längere Sequenzen in der Mixfixnotation $\langle 1, 7, 3, 12, 9\rangle$ zu schreiben.

[3] Für die Benennung dieses Typs gibt es verschiedene Traditionen in der Literatur. Wir werden hier sowohl *Seq* α als auch *List* α verwenden.

Für Sequenzen hat man gerne eine multiple Konstruktorsicht, je nachdem, ob man sie von links nach rechts oder von rechts nach links verarbeiten will. Dabei drückt das Schlüsselwort GENERATED aus, dass die jeweils angegebenen Funktionen ausreichen, um alle Elemente des Typs zu erzeugen.[4]

PROP *Seq* GENERATED BY $\Diamond$, $\cdot$: $--$ *echter Konstruktor mit prepend*
PROP *Seq* GENERATED BY $\Diamond$, $:\cdot$ $--$ *Pseudokonstruktor mit append*

Die zugehörigen Matching-Propertys sehen dann so aus:

PROP s MATCHES $(lead :\cdot z) \iff s \neq \Diamond$
PROP s MATCHES $(lead :\cdot z) \implies lead = front(s) \wedge z = last(s)$

Man beachte, dass die Definitionen auf Basis dieser Pseudokonstruktoren im Allgemeinen extrem ineffizient sind, weil bei einem Pattern der Bauart

DEF $f(lead :\cdot x) = \ldots f(lead) \ldots x \ldots$

das Element x nur mit dem Aufwand $\mathcal{O}(n)$ erreicht werden kann, was insgesamt wegen der Rekursion zu einem Aufwand der Größenordnung $\mathcal{O}(n^2)$ führt. (In Kapitel 12 werden wir dazu allerdings noch Erfreuliches erfahren.)

In Abschnitt 1.2.2 werden wir gleich noch eine Reihe weiterer fundamentaler Operationen auf Listen kennen lernen, die unter dem Schlagwort *Map-Filter-Reduce* bekannt geworden sind.

Einige Sprachen, z. B. HASKELL, führen für den Listentyp spezielle Notationen ein: Mit $[Nat]$ wird der Typ der Listen über *Nat* beschrieben, was unserem *Seq Nat* entspricht. Außerdem gibt es noch spezielle Schreibweisen für Listenkomprehension, endliche Listen etc.

1.2 Funktionale

Zum Kern der Funktionalen Programmierung gehört das Konzept der *Funktionen höherer Ordnung* oder der *Funktionale*. Durch sie kommt ein großer Teil der Eleganz, der Kompaktheit und damit der Produktivität dieses Programmierparadigmas zustande.

Während man in der Frühzeit der Programmierung noch die Befürchtung hatte, dass die Eleganz von Funktionalen mit hohen Effizienzverlusten bezahlt werden muss, weiß man heute, dass die Compiler solche Funktionen nahezu ohne Overhead implementieren können. Es gibt also keinen Grund mehr, darauf zu verzichten.

[4] Dieses so genannte *Erzeugungsprinzip* ist eine fundamentale Eigenschaft sowohl in der algebraischen Spezifikation von Datentypen als auch in Beweissystemen.

Definition (Funktional, Funktion höherer Ordnung)

Funktionen, die als Parameter oder Resultate wieder Funktionen haben, bezeichnet man als ***Funktionen höherer Ordnung*** bzw. ***Funktionale***.

Ist z. B. eine reelle Funktion gegeben, so kann man sie verschieben, spiegeln oder strecken (vgl. Abbildung 1.1). Das führt zu folgenden Funktionen:

FUN *shift*: $Real \to (Real \to Real) \to (Real \to Real)$
DEF $shift\ dx\ f\ x = f(x - dx)$

FUN *mirror*: $(Real \to Real) \to (Real \to Real)$
DEF $mirror\ f\ x = f(-x)$

FUN *stretch*: $Real \to (Real \to Real) \to (Real \to Real)$
DEF $stretch\ r\ f\ x = f(\frac{x}{r})$

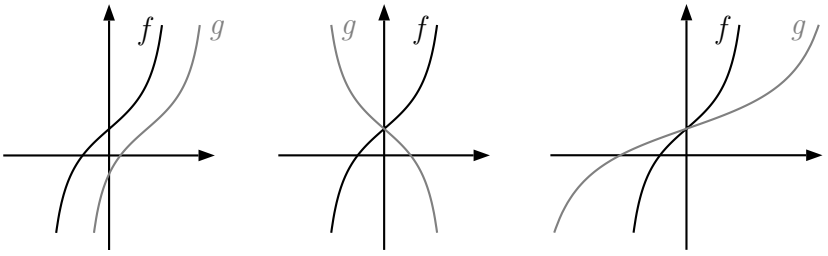


Abb. 1.1: Anwendung von Funktionalen auf eine Funktion f

Für Funktionen, die Resultate einer Berechnung sind, benutzt man auch gerne die λ-Notation:

FUN *shift*: $Real \to (Real \to Real) \to (Real \to Real)$
DEF $shift\ dx\ f = \lambda x \bullet f(x - dx)$

Dies drückt deutlich aus, dass z. B. durch die Applikation $shift(\frac{\pi}{2})\ cos$ eine Funktion erzeugt wird, deren Verlauf der Cosinus-Funktion entspricht, die aber um $\frac{\pi}{2}$ auf der x-Achse nach rechts verschoben ist.

Wir setzen im Folgenden dieses Konzept als bekannt voraus und erinnern hier nur an einige wichtige Funktionale, die wir im weiteren Verlauf immer wieder verwenden werden.

1.2.1 Allgemeine Funktionale

Es gibt eine Vielzahl von Funktionen höherer Ordnung, die aus der Mathematik wohl bekannt sind und deren Verfügbarkeit viele Programme eleganter

schreiben lässt. Einige repräsentative Vertreter dieser Funktionsfamilie haben wir in einer so genannten *Struktur* (vgl. Kapitel 4) in Programm 1.1 zusammengefasst. Man beachte, dass alle diese Funktionen polymorph sind.

Programm 1.1 Nützliche Funktionen höherer Ordnung

```
STRUCTURE HigherOrder = {
  FUN _ . _: α → (α → β) → β                          -- Applikation, invers
  DEF x.f = f x
  FUN _ ∘ _:  (β → γ) → (α → β) → α → γ               -- Funktionskomposition
  DEF (g ∘ f) x = g (f x)
  FUN _ ; _:  (α → β) → (β → γ) → α → γ               -- Funktionskomposition
  DEF (f ; g) = g ∘ f
  FUN id:  α → α                                      -- Identität
  DEF id x = x
  FUN K:  α → β → α                                   -- Konstante Funktion
  DEF K a b = a
  FUN curry:  (α × β → γ) → (α → β → γ)               -- Currying
  DEF (curry f) a b = f(a, b)
  FUN uncurry:  (α → β → γ) → (α × β → γ)             -- Uncurrying
  DEF (uncurry g)(a, b) = g a b
  FUN _ ^ _:  (α → α) × Nat → (α → α)                 -- Iterierte Applikation
  DEF f^0 = id                                        -- geschrieben f^n
  DEF f^n = f^(n−1) ∘ f
  FUN _ while _:  (α → α) × (α → Bool) → (α → α)      -- Wiederholung
  DEF (f while p) x = IF p x THEN (f while p)(f x) ELSE x FI
}
```

Mit der invertierten Funktionsapplikation lässt sich eine Notation simulieren, die in der objektorientierten Programmierung sehr beliebt ist (und manchmal sogar als Konzept verkauft wird). Wir werden davon in Kapitel 4 extensiv Gebrauch machen, um unsere konzeptuelle Funktionssicht mit bekannten traditionellen Notationen in Einklang zu bringen.

Wir führen noch eine Konvention ein, die in einigen Anwendungen zu kompakten und eleganten Notationen führt. Sei eine Menge $f_1 : \alpha \to \beta_1, \ldots, f_n : \alpha \to \beta_n$ von Funktionen über dem gleichen Definitionsbereich gegeben. Dann lassen wir die *Tupelapplikation* zu:

$$(f_1, \ldots, f_n)(a) = (f_1(a), \ldots, f_n(a)) \quad -- \textit{ Tupelapplikation}$$

1.2.2 Catamorphismen (Map-Filter-Reduce)

Sehr viele elegante Algorithmen der Funktionalen Programmierung basieren auf dem so genannten *Map-Filter-Reduce*-Prinzip. Dahinter steckt die Beob-

achtung, dass die Algorithmen genau den Aufbau der zugrunde liegenden Datenstruktur widerspiegeln. Man spricht daher auch vom *Homomorphie-Prinzip* oder – wenn man Eindruck schinden will – von *Catamorphismen*. Diese Programmiertechnik ist in den letzten Jahren sehr stark von Richard Bird und Lambert Meertens ausgebaut worden. Genaueres findet man z. B. in [20], wo auch weitere Literaturhinweise stehen.

Anmerkung: Jeffrey Dean und Sanjay Ghemawat von der Firma Google beschreiben in [40] ein so genanntes MapReduce-Pattern, das in ihrer Firma in Hunderten von Programmen implementiert wurde, die täglich in mehr als Tausend Jobs viele Terabytes von Daten verarbeiten. Das ist ein deutliches Indiz für die zentrale Bedeutung dieses Paradigmas. Es stützt auch die These, dass dieses Muster in ungezählten PASCAL-, C- *oder* JAVA-*Programmen steckt – wenn auch unerkannt.*

Catamorphismen lassen sich im Prinzip auf allen möglichen Datenstrukturen in analoger Weise definieren. Wir betrachten sie hier exemplarisch für den Fall der endlichen Sequenzen (vgl. Abschnitt 1.1.11). In Kapitel 2 werden wir das Konzept auf unendliche Listen erweitern.

Die Map-Operation

Die *Map*-Operation wendet eine Funktion f auf alle Elemente einer Sequenz an.

FUN $map\colon (\alpha \to \beta) \to Seq\ \alpha \to Seq\ \beta$ $\quad -- \ Map\text{-}Funktion$
DEF $map\ f\ \Diamond = \Diamond$
DEF $map\ f\ (a\ \cdot\colon rest) = (f\ a)\ \cdot\colon (map\ f\ rest)$

Statt *map* werden wir meist „ $*$ “ schreiben und dabei faktisch eine Infix-Notation verwenden, die allerdings etwas trickreich realisiert ist, indem wir die Operation „ $*$ “ als Postfix-Operation mit Currying definieren. Der Vorteil ist, dass wir dann Terme wie $(f*)$ von vornherein als legale Konstrukte haben und Dinge wie $(g*) \circ (f*)$ schreiben dürfen.

FUN $\_*\colon\ (\alpha \to \beta) \to Seq\ \alpha \to Seq\ \beta$ $\quad -- \ alternative\ Schreibweise$
DEF $f * s = map\ f\ s$

Die Map-Operation lässt sich naheliegend auf mehrstellige Funktionen verallgemeinern. Wir erlauben also generell z. B. zu schreiben

$$\ldots + *(a,\ b) \ldots$$

was bedeutet, dass die beiden Sequenzen a und b elementweise addiert werden. Das lässt sich ebenso mit drei-, vier-, fünf-stelligen Operationen etc. machen. Wir verzichten darauf, das hier einzeln zu definieren.

Ein Problem muss allerdings geklärt werden: *Was geschieht bei verschieden langen Sequenzen?* Um die Undefiniertheiten möglichst gering zu halten, legen wir fest, dass in diesen Fällen das Ergebnis die Länge der *kürzesten* Argumentsequenz hat.

FUN map: $(\alpha_1 \times \alpha_2 \to \beta) \to (Seq\ \alpha_1 \times Seq\ \alpha_2 \to Seq\ \beta)$
DEF $map\ f\ (\Diamond, \Diamond) = \Diamond$
DEF $map\ f\ (\Diamond, s_2) = \Diamond$
DEF $map\ f\ (s_1, \Diamond) = \Diamond$
DEF $map\ f\ (a_1 \cdot\!: rest_1, a_2 \cdot\!: rest_2) = f(a_1, a_2) \cdot\!: map\ f\ (rest_1, rest_2)$

Den ersten Fall hätten wir weglassen können, weil er von den nächsten beiden jeweils mit abgedeckt wird. Die vollständige Definition ist aber klarer und deshalb besser.

Die Zip-Operation

Die Verallgemeinerung auf mehrstellige Funktionen macht die Map-Operation sehr flexibel. Aber für den zweistelligen Fall möchte man gerne die Infix-Schreibweise retten. Dafür hat sich in der Literatur der Name *Zip* eingebürgert, da die beiden Sequenzen „reißverschlussartig“ zusammengefügt werden.

FUN $\_ \setminus \_ / \_$: $Seq\ \alpha \times (\alpha \times \beta \to \gamma) \times Seq\ \beta \to Seq\ \gamma$ −− *Zip*
DEF $a \setminus \oplus / b = \oplus * (a, b)$ −− *geschrieben* $a \overset{\oplus}{\curlyvee} b$

Man beachte, dass bei verschieden langen Sequenzen die längere abgeschnitten wird; wir haben also insbesondere z. B. die Eigenschaft $\Diamond \overset{\oplus}{\curlyvee} b = \Diamond$.

Die Filter-Operation

Die *Filter*-Operation extrahiert aus einer Sequenz die Teilsequenz derjenigen Elemente, die ein gegebenes Prädikat p erfüllen.

FUN *filter*: $(\alpha \to Bool) \to Seq\ \alpha \to Seq\ \alpha$ −− *Filter-Funktion*
DEF *filter* $p\ \Diamond = \Diamond$
DEF *filter* $p\ (a \cdot\!: rest) =$ IF $p\ a$ THEN $a \cdot\!:$ *filter* p *rest*
ELSE *filter* p *rest* FI

Aus Gründen der Flexibilität geben wir zwei notationelle Varianten an, wobei wir bei der ersten wieder von Infix- auf Postfix-Notation übergehen, um z. B. Dinge wie $(g*) \circ (p\triangleleft) \circ (f*)$ schreiben zu können. Man beachte, dass wir dabei trotzdem die Notation $p \triangleleft s$ beibehalten können, die zumindest wie eine Infix-Notation aussieht.

FUN $\_\triangleleft$: $(\alpha \to Bool) \to Seq\ \alpha \to Seq\ \alpha$ −− *Filter-Funktion*
DEF $p \triangleleft s = filter\ p\ s$
FUN $\_ \triangleright \_$: $Seq\ \alpha \times (\alpha \to Bool) \to Seq\ \alpha$
DEF $s \triangleright p = p \triangleleft s$

Anmerkung: Diese Funktion lässt sich nicht *ohne weiteres auf Strukturen wie Bäume oder Matrizen übertragen, da sie die Gestalt der Datenstruktur zerstören kann. Um dieses Problem zu umgehen, führt man für die dann fehlenden Elemente „virtuelle“ Platzhalter ein, z. B. mit Hilfe der Datenstrukturen option in* OPAL *oder Maybe in* HASKELL *(s. Kapitel 8).*

Für gewisse Aufgaben brauchen wir weitere Varianten der Funktionsfamilie Filter, die mit zwei- statt mit einstelligen Prädikaten arbeiten. Im Wesentlichen geht es darum, die Sequenzelemente relativ zu ihren Nachbarn zu bewerten. Während aber das Filtern mit einstelligen Prädikaten offensichtlich ist, muss man sich bei zweistelligen Prädikaten zwischen mehreren möglichen Festlegungen entscheiden. Wir können z. B. bestimmen, dass $(<) \triangleleft s$ diejenigen Elemente beibehalten soll, die jeweils kleiner als ihr Nachfolger sind:

$$(<) \triangleleft \langle 2,5,1,1,6,6,4,7\rangle = \langle 2,1,4\rangle$$

Diese Funktion kann folgendermaßen definiert werden:

```
FUN _◁: (α × α → Bool) → Seq α → Seq α          -- Filter-Funktion
DEF p ◁ ◊ = ◊
DEF p ◁ ⟨x⟩ = ◊
DEF p ◁ (x ·: y ·: rest) = IF p(x, y) THEN x ·: (p ◁ (y ·: rest))
                                      ELSE      (p ◁ (y ·: rest)) FI
FUN _ ▷ _: Seq α × (α × α → Bool) → Seq α
DEF s ▷ p = p ◁ s
```

Die Reduce-Operation

Schließlich betrachten wir noch die *Reduce*-Funktion, die die Elemente einer Sequenz mit Hilfe einer Operation „akkumuliert“. Klassische Beispiele sind die Summe aller Elemente, das Produkt aller Elemente etc. Diese Funktion muss allerdings in mehreren Varianten bereitgestellt werden, da die leere Sequenz bzw. fehlende Assoziativität Probleme machen können. Wir verwenden wieder eine Postfix- anstelle der Infix-Notation.

```
FUN _/:  (α × α → α) → Seq α → α     -- Reduce
FUN _↚:  (α × α → α) → Seq α → α     -- alternative Notation
DEF ↚ = /
DEF ⊕ /⟨a⟩ = a
DEF ⊕ /(a ·: rest) = a ⊕ (⊕ / rest)
```

Diese Funktion führt z. B. zu folgender Auswertung: $+ /\langle a,b,c,d,e\rangle = a + (b + (c + (d + e)))$. Deshalb verwenden wir, wenn wir die Richtung der Auswertung verdeutlichen wollen, auch das Symbol $\not\leftarrow$; das heißt, $\oplus /s \mathrel{\widehat{=}} \oplus \not\leftarrow s$. Das mag beim Minuszeichen anstelle von Plus nicht gewünscht sein. Deshalb gibt es auch einen Operator für die inverse Richtung.[5]

```
FUN _ ↛:  (α × α → α) → Seq α → α
DEF ⊕ ↛⟨a⟩ = a
DEF ⊕ ↛(lead ·: a) = (⊕ ↛ lead) ⊕ a
```

[5] Man beachte, dass wir hierbei erweitertes Patternmatching mittels der Funktion ·: verwenden, die ein Element „hinten“ an eine Sequenz anhängt.

Mit diesen Funktionen kann man schnell Algorithmen zusammensetzen. Ein typisches Beispiel ist die Standardabweichung einer Verteilung, die definiert ist als die Wurzel der Summe der Quadrate aller Abweichungen vom Mittelwert m:

$$\text{DEF } abw\ s = \sqrt{+ / (\_\ \hat{}\ 2) * |m - \_\ | * s} \quad \text{WHERE } m = mittel(s)$$
$$\text{DEF } mittel\ s = \frac{+/s}{length\ s}$$

Ein Problem bleibt: Die Reduce-Operatoren sind für die leere Sequenz nicht definiert. Für dieses Problem bieten sich zwei Lösungen an: eine konventionelle und eine fortschrittliche. Wir beginnen mit der fortschrittlichen.

Motiviert durch die klassischen mathematischen Definitionen, die z. B. die Summe über der leeren Menge als 0 und das Produkt über der leeren Menge als 1 festlegen, wählen wir als Standardwert für $\oplus / \Diamond$ das *neutrale Element* der Operation $\oplus$. Das heißt, wir definieren die Reduce-Operation nicht über beliebigen Typen α und Operationen $\oplus$, sondern nur über solchen, die ein neutrales Element besitzen. Dazu benötigen wir das Sprachmittel der Typklassen, das in Kapitel 9 eingeführt wird. Mit den dort beschriebenen Notationen können wir Reduce folgendermaßen definieren:

$$\text{FUN } \_/\colon (\alpha \times \alpha \to \alpha) \to Seq\ \alpha \to \alpha \quad \text{VAR } \alpha\colon \mathit{Monoid} \quad -- \ Reduce$$
$$\text{DEF } \oplus / \Diamond = unit \qquad -- \ neutrales\ Element$$
$$\text{DEF } \oplus / (a \cdot\colon rest) = a \oplus (\oplus / rest)$$

In der Typklasse *Monoid* wird ein neutrales Element *unit* eingeführt. Und die Struktur *Nat* kann als Monoid bzgl. $(+, 0)$ und auch bzgl. $(\cdot, 1)$ charakterisiert werden. Entsprechendes gilt z. B. auch für Sequenzen *Seq* bzgl. $(+\!+, \Diamond)$ oder für Mengen *Set* bzgl. $(\cup, \emptyset)$ etc. (Mehr dazu in Kapitel 9.)

Die konventionelle Lösung besteht darin, das fehlende Element als zusätzliches Argument bereitzustellen, und zwar entweder über Parametertupel, über Currying oder über Mixfix-Notationen. (Wir wählen letztere Variante.)

Als zusätzlichen Effekt können wir jetzt die Operation etwas verallgemeinern: Das abschließende Element braucht nicht das neutrale Element zu sein und es kann sogar einen anderen Typ β haben (sofern $\oplus$ dazu passt).

$$\text{FUN } \_ / \_ \mid \_\colon (\alpha \times \beta \to \beta) \times Seq\ \alpha \times \beta \to \beta$$
$$\text{DEF } \oplus / \Diamond \mid e = e$$
$$\text{DEF } \oplus / a \cdot\colon rest \mid e = a \oplus (\oplus / rest \mid e)$$

In der BIBLIOTHECA OPALICA [1], also der Bibliothek der Sprache OPAL, wird das etwas anders geschrieben: $(\oplus, e)\ /\ S$.

Der generelle Hom-Operator

Alle in diesem Abschnitt eingeführten Funktionen können über eine einzige Funktion mit entsprechend vielen Parametern definiert werden:

$$\begin{array}{l}\text{FUN } hom\colon\ (\gamma \times \delta \to \delta) \times \delta \times (\beta \to \gamma) \times (\beta \to Bool) \times (\alpha \to \beta) \\ \qquad\qquad \to (Seq\ \alpha \to \delta) \\ \text{DEF } hom(\oplus,\ e,\ g,\ p,\ f)(\Diamond) = e \\ \text{DEF } hom(\oplus,\ e,\ g,\ p,\ f)(a\ \cdot\!:\ rest) = \\ \qquad \text{IF } p\ (f\ \ a) \text{ THEN } (g\ (f\ \ a)) \oplus hom(\oplus,\ e,\ g,\ p,\ f)(rest) \\ \qquad\qquad\quad \text{ELSE} \qquad\qquad\quad\ hom(\oplus,\ e,\ g,\ p,\ f)(rest) \text{ FI}\end{array}$$

Damit bestehen folgende Gleichungen (in denen wir einige der Funktionale aus Programm 1.1 auf Seite 25 verwenden):

$$\begin{array}{l}\text{PROP } f * s = hom(\cdot\!:,\ \Diamond,\ id,\ K\ true,\ f)(s) \\ \text{PROP } p \lhd s = hom(\cdot\!:,\ \Diamond,\ id,\ p,\ id)(s) \\ \text{PROP } \ \oplus /s \mid e = hom(\oplus,\ e,\ id,\ K\ true,\ id)(s)\end{array}$$

Umgekehrt kann auch *hom* als Komposition der anderen Operatoren definiert werden:

$$\text{PROP } hom(\oplus,\ e,\ g,\ p,\ f)(s) = \oplus\ /\ g * p \lhd f * s \mid e$$

Mit der Funktion *hom* kann unser obiges Beispiel der mittleren Abweichung so geschrieben werden:

$$\text{DEF } abw\ s = \sqrt{hom(+,\ 0,\ id,\ K\ true,\ (\_\ \hat{}\ 2) \circ |m - \_|)(s)} \quad \text{WHERE} \ldots$$

Dabei haben wir noch eine weitere der elementaren Gleichungen für das Map-Filter-Reduce-Paradigma benutzt:

$$\begin{array}{l}\text{PROP } g * f * s = (g \circ f) * s \\ \text{PROP } p \lhd f * s = (p \circ f) \lhd s\end{array}$$

Beispiel 1.1

Zum Abschluss dieser einführenden Erinnerung soll noch das Standardbeispiel *quicksort* zeigen, wie diese Funktionale eingesetzt werden können:

$$\begin{array}{l}\text{FUN } quicksort\colon\ Seq\ \alpha \to Seq\ \alpha \\ \text{DEF } quicksort\ \Diamond = \Diamond \\ \text{DEF } quicksort\ s = \text{LET } x \quad = \ arb\ s \\ \qquad\qquad\qquad\qquad\quad\ small \ = \ (< x) \lhd s \\ \qquad\qquad\qquad\qquad\quad\ medium \ = \ (= x) \lhd s \\ \qquad\qquad\qquad\qquad\quad\ large \ = \ (> x) \lhd s \\ \qquad\qquad\qquad\quad \text{IN} \\ \qquad\qquad\qquad\qquad\quad (quicksort\ small) +\!\!+ medium +\!\!+ (quicksort\ large)\end{array}$$

1.3 Semantik und Auswertungsstrategien

Bei Sprachen unterscheidet man generell zwischen ihrer ***Syntax***, also der *Schreibweise* der einzelnen Sprachkonstrukte, und ihrer ***Semantik***, also der *Bedeutung* der einzelnen Sprachkonstrukte.

Bei funktionalen Sprachen sollte man erwarten, dass die Definition der Semantik trivial ist, weil sie sich direkt auf den mathematischen Funktionsbegriff stützt. Im Prinzip ist das auch so, aber es gibt doch ein paar Subtilitäten, die wir kurz ansprechen müssen. Wir können diese grundlegenden Fragen der Semantik hier nur skizzieren; für weitergehende Informationen verweisen wir auf Spezialliteratur, z. B. [135, 60, 126, 63, 146]. Für die speziellen Aspekte, die wir hier benötigen, sei auch auf den nach wie vor lesenswerten (und zum Glück wieder erhältlichen) Klassiker von Zohar Manna [96] verwiesen; auch in [50] findet man eine detaillierte Diskussion.

Im Rahmen dieses Buches sind vor allem zwei semantische Fragen von Bedeutung, weil die verschiedenen Programmiersprachen und -techniken sich in diesen beiden Punkten auf eine für das Programmieren relevante Weise unterscheiden.

- Wie werden die Argumente von Funktionen behandelt?
- Wie wird Rekursion behandelt?

Diese beiden Aspekte wollen wir im Folgenden kurz betrachten. Wir tun das im Rahmen der zwei wichtigsten Arten der Semantikdefinition von Sprachen, der so genannten *denotationellen Semantik* und der so genannten *operationalen Semantik*.

1.3.1 Denotationelle Semantik

Bei der ***denotationellen Semantik*** wird jedes Programm direkt als („Denotation“ für) eine mathematische Funktion charakterisiert.

Der kritischste Punkt ist dabei der Umgang mit ***partiellen Funktionen***. In der traditionellen Mathematik werden solche Funktionen gerne schamhaft „wegdefiniert“, aber in der Informatik kann man sich das nicht leisten. Fehlerhafte Operationen wie die Division $\frac{x}{y}$ für $y = 0$ oder nichtterminierende Programme wie

$\text{DEF } f(0) = 1$
$\text{DEF } f(x) = x \cdot f(x+1)$

sind Phänomene, die sich nicht ignorieren lassen.

Um solche Situationen auf der semantischen Ebene in den Griff zu bekommen, führt man ein spezielles Element „$\perp$“ (genannt „Bottom“) ein, das für „undefiniert“ steht. Statt zu sagen, *„die Funktion f ist an der Stelle a undefiniert“*, kann man dann einfach schreiben „$f(a) = \perp$“.

Anmerkung: Man beachte, dass das nur eine bequeme Kurznotation ist, kein syntaktisches Konstrukt der Programmiersprache. Ausdrücke wie IF $x = \perp$ THEN ... *sind in Programmen* ***nicht*** *zulässig!*

In diesem Zusammenhang betrifft eine wesentliche Designentscheidung bei Programmiersprachen die Frage, wie undefinierte Argumente von Funktionen behandelt werden. Dabei unterscheidet man zwei Arten von Funktionen: strikte und nichtstrikte.

Definition (Strikte und nichtstrikte Funktionen)

- ***Strikte Funktionen*** sind undefiniert, sobald mindestens eines der Argumente undefiniert ist. Kurz

  $$f(\ldots, \bot, \ldots) = \bot$$

- ***Nichtstrikte Funktionen*** können auch dann noch definiert sein, wenn eines ihrer Argumente undefiniert ist.

  $$f(\ldots, \bot, \ldots) \neq \bot \quad \text{(abhängig von den übrigen Argumenten)}$$

Bei Bedarf verwendet man auch eine präzisere Terminologie und sagt, *die Funktion f ist strikt (bzw. nichtstrikt) im i-ten Argument.*

Die große Mehrheit der Funktionen, die wir in der Programmierung verwenden, sind strikt. Aber es gibt auch bedeutende Ausnahmen: Das IF.THEN.ELSE.FI-Konstrukt kann als Funktion *if*(_ , _ , _) mit drei Argumenten aufgefasst werden. Diese Funktion ist strikt im ersten und nichtstrikt im zweiten und dritten Argument.[6]

In den meisten Programmiersprachen sind die benutzerdefinierten Funktionen grundsätzlich strikt; das gilt in nahezu allen imperativen Sprachen, aber auch in funktionalen Sprachen wie ML oder OPAL. In einigen wenigen Sprachen sind diese Funktionen dagegen nichtstrikt; das prominenteste Beispiel dafür ist HASKELL. Es gibt aber auch nichtstrikte ML-Dialekte.

Für die semantische Erklärung *rekursiver Deklarationen* wird dann eine spezielle partielle Ordnung „$\sqsubseteq$“ („*less defined*“) eingeführt, in der „$\bot$“ das kleinste Element ist. Über dieser Ordnung werden rekursive Deklarationen als so genannte *kleinste Fixpunkte* der zugehörigen Gleichungen interpretiert. Wir kommen auf diese Konstruktion in Kapitel 10 im Zusammenhang mit Fixpunktprogrammierung noch ausgiebiger zurück.

Die denotationelle Semantik ist vor allem dann relevant, wenn man „mit Programmen rechnen“ will (vgl. Abschnitt 1.4) und deshalb die gültigen Rechenregeln kennen muss.

1.3.2 Operationale Semantik

Bei der ***operationalen Semantik*** definiert man, wie die Programme auf einer geeignet konzipierten *abstrakten Maschine* ausgewertet werden. Die Menge

[6] Man kann zeigen, dass jede Programmiersprache mit Rekursion oder Iteration (also jede praktisch brauchbare Sprache) mindestens ein nichtstriktes Konstrukt enthalten muss. Meistens übernimmt das *if* diese Rolle.

aller Auswertungen induziert dann eine Funktion (die mit der denotationellen Semantik identisch sein sollte).

Auch in diesem Rahmen ist die Frage der Argumentauswertung ein zentraler Aspekt der Semantik; denn strikte und nichtstrikte Funktionen müssen sich unterscheiden. Allerdings zeigt sich, dass die operationale Semantik filigraner ist: Bei den nichtstrikten Funktionen gibt es unterschiedliche Varianten.

Die Essenz einer operationalen Semantik manifestiert sich in den so genannten *Auswertungsstrategien*, also der Art, wie die Maschine Funktionsargumente behandelt. In unserem Kontext sind vor allem drei Strategien von Bedeutung: Call-by-value, Call-by-name und Call-by-need.

Definition (Call-by-value, Call-by-name und Call-by-need)

- ***Call-by-value*** (auch *eager* genannt). Bei dieser Strategie werden alle Argumentausdrücke vollständig ausgewertet, bevor die Funktion (mit diesen Argument*werten*) aufgerufen wird. Das geschieht unabhängig davon, ob das Argument zur Berechnung des Funktionswerts überhaupt gebraucht wird oder nicht. Funktionen, die mit Call-by-value ausgewertet werden, sind *strikt*.
- ***Call-by-name***. Hier werden beim Funktionsaufruf die Argument*ausdrücke* selbst unausgewertet übergeben und erst bei der Verwendung der entsprechenden Parameter im Rumpf berechnet.[7] Funktionen, die mit Call-by-name ausgewertet werden, sind *nichtstrikt*.
- ***Call-by-need*** (auch ***lazy*** genannt). Das ist eine Variante der Call-by-name-Strategie, bei der durch so genanntes *Sharing* verhindert wird, dass die Argumentausdrücke unter Umständen mehrfach ausgerechnet werden. Funktionen, die mit Call-by-need ausgewertet werden, sind *nichtstrikt*.

Um die Unterschiede dieser Strategien zu illustrieren, verwenden wir ein einfaches, wenn auch etwas artifizielles Beispiel.

$\text{DEF } foo(x, y) = \text{IF } x > 0 \text{ THEN } x \text{ ELSE } y \text{ FI}$

Betrachten wir folgenden Aufruf:

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$

Bei der *Call-by-value*-Strategie werden zuerst die Argumente ausgewertet, was auf $foo(1, \frac{1}{0})$ führt. Weil $\frac{1}{0}$ undefiniert ist, scheitert die Auswertung des zweiten Arguments, wodurch der ganze Aufruf undefiniert ist. (Es wird ein „Zero-divide-Error“ gemeldet.) Das ist umso ärgerlicher, weil für $x = 1$ der Parameter y gar nicht gebraucht wird. Insgesamt haben wir also die folgende Auswertung:

[7] Manchmal unterscheidet man noch genauer zwischen *Call-by-name* und *Call-by-expression*; aber in unserer Betrachtung genügt diese etwas gröbere Sichtweise.

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$ $\quad --$ *call-by-value*
$\rightsquigarrow foo(1, \frac{1}{0})$
$\rightsquigarrow foo(1, \bot)$
$\rightsquigarrow \bot$

Bei der *Call-by-name*-Strategie werden die beiden Argumentausdrücke vorläufig nicht ausgewertet. Erst wenn die entsprechenden Parameter im Rumpf benötigt werden, erfolgt die Berechnung der Argumente. Das führt in unserem Beispiel auf folgende Auswertung:

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$ $\quad --$ *call-by-name*
$\rightsquigarrow$ IF $cos(2\pi) > 0$ THEN $cos(2\pi)$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow$ IF $1 > 0$ THEN $cos(2\pi)$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow cos(2\pi)$
$\rightsquigarrow 1$

Weil für den gegebenen x-Wert der y-Wert nie gebraucht wird, hat die Undefiniertheit von $\frac{1}{sin(2\pi)}$ keine negative Auswirkung. Aber man erkennt in diesem Beispiel auch den gravierenden Nachteil der Call-by-name-Strategie: Der Ausdruck $cos(2\pi)$ wird zweimal ausgewertet!

Die *Call-by-need*-Strategie will genau dieses Effizienzproblem beheben. Indem der Compiler intern geeignete Pointer setzt, erreicht man so genanntes *Sharing*. Das heißt, sobald der Argumentwert das erste Mal berechnet wurde, steht er auch allen anderen Applikationsstellen zur Verfügung. Damit sieht unsere Auswertung folgendermaßen aus:

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$ $\quad --$ *call-by-need (lazy)*
$\rightsquigarrow$ IF $\boxed{cos(2\pi)} > 0$ THEN $\boxed{cos(2\pi)}$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow$ IF $\boxed{1} > 0$ THEN $\boxed{1}$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow 1$

Die Call-by-need-Auswertung ist zwar effizienter als die Call-by-name-Auswertung; aber mit der Call-by-value-Auswertung kann sie trotzdem nicht mithalten. Deshalb werden Sprachendesigner, die auf Effizienz Wert legen, immer eine Call-by-value-Semantik vorziehen.[8] In Kapitel 2 werden wir allerdings sehen, dass Laziness ein wichtiges Hilfsmittel ist, um *unendliche Datenstrukturen* zu implementieren. Dort werden wir auch zeigen, wie sich „punktuelle“ Laziness in eine Call-by-value-Sprache einbauen lässt.

Im Folgenden werden wir – wie bei funktionalen Sprachen üblich – die Begriffe *Call-by-value* und *strikt* sowie *Call-by-need* und *lazy* jeweils synonym verwenden.

[8] Im Compilerbau wird die Technik der so genannten Striktheitsanalyse eingesetzt, um bei Call-by-name- oder Call-by-need-Sprachen den Overhead möglichst klein zu halten. Aber diese Technik ist aufwendig und findet nicht alle Optimierungen.

Anmerkung 1: In manchen Büchern, z. B. [50], wird bei Laziness und Call-by-need filigraner unterschieden. Laziness bedeutet dort, dass die Argumente von Konstruktor*funktionen nicht ausgewertet werden; das reicht aus, um unendliche Datenstrukturen zu implementieren.*

Anmerkung 2: Die Auswertungsstrategien funktionaler Sprachen stehen in engem Zusammenhang mit den Reduktionsstrategien von λ-Ausdrücken im λ-Kalkül. So korrespondiert die Call-by-value-Strategie zur so genannten Applicative-order-reduction. Die Call-by-name-Strategie entspricht der Normal-order-reduction. Genauere Betrachtungen hierzu findet man in [50].

1.4 Mit Programmen rechnen

Funktionale Programme sind semantisch direkt im mathematischen Funktionsbegriff verankert. Das hat einen außerordentlich angenehmen Nebeneffekt: Man kann mit den Programmen genauso rechnen, wie man es an anderen Stellen in der Mathematik gelernt hat. Dabei gibt es durchaus Analogien zum Vorgehen eines Ingenieurs, der eine Differenzialgleichung lösen will: Er kennt eine Reihe von Standardansätzen, die er der Reihe nach probiert, bis einer funktioniert. (Wenn keiner klappt, ist das Problem wirklich schwer.)

Anmerkung: Dieser Ansatz ist ein Zweig der Programmierung, der unter Begriffen wie „Programmtransformation“ oder „deduktive Programmierung“ bekannt geworden ist. Eine umfassende Darstellung findet sich z. B. in [18, 109]. Besonders weit wurde diese Methode von Bird und Meertens entwickelt, vor allem im Zusammenhang mit Map-Filter-Reduce-Programmen [21].

Dabei gibt es zwei prinzipielle Vorgehensweisen. Man kann die Regeln sehr rigoros und formal fassen (als so genannte *Transformationsregeln*), so dass sie weitgehend automatisch von entsprechenden Werkzeugen – im Idealfall vom Compiler – angewandt werden können. Oder man betrachtet das Ganze eher als eine Methode, mit der man seine Programme selbst weiterentwickeln kann. Dabei fängt man oft mit einer (nicht-ausführbaren) Spezifikation an, leitet daraus eine elegante und korrekte aber nicht besonders effiziente erste Lösung ab und entwickelt diese schließlich weiter in ein weniger elegantes aber dafür effizientes Programm. Wir werden uns im Folgenden an diese zweite Sicht halten.

Im Rahmen dieses Buches müssen wir uns auf eine exemplarische Skizze dieser Methodik beschränken. Dabei konzentrieren wir uns auf diejenigen Aspekte, die in späteren Kapiteln noch benötigt werden.

1.4.1 Von linearer Rekursion zu Tail-Rekursion

Ein wichtiges Effizienzproblem bei funktionalen Sprachen betrifft die *effiziente Ausführung rekursiver Funktionen*. In imperativen Sprachen ist das kein so wichtiges Thema, weil man dort primär Schleifen programmiert, die a priori effizient sind (dafür aber weniger elegant und fehleranfälliger).

Die einfachste Form von Rekursion ist die so genannte ***Tail-Rekursion***.[9] Bei dieser Form ist der rekursive Aufruf die äußerste („letzte“) Operation im jeweiligen Zweig der Fallunterscheidung. Ein typisches Beispiel ist die Berechnung des Rests bei der Division:

$$\begin{array}{l}\text{FUN } mod\colon Nat \times Nat \to Nat\\ \text{DEF } mod(a, b) = \text{IF } a < b \text{ THEN } a\\ \qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } mod(a - b,\ b) \text{ FI} \quad -- \textit{Tail-Rekursion}\end{array}$$

Bei musterbasierten Definitionen ist das noch offensichtlicher: dort ist der rekursive Aufruf ganz außen.

Solche Tail-rekursiven Funktionen sind äußerst effizient. Jeder halbwegs ordentliche Compiler erkennt heute diese spezielle Rekursionsform und setzt sie im Maschinencode unmittelbar in Schleifen bzw. direkt in Gotos um.

Aus diesem Grund ist es interessant, funktionale Programme weitestgehend aus Tail-rekursiven Funktionen zusammenzusetzen. Leider ist diese Form aber in vielen Situationen nicht die eleganteste oder „natürlichste“ Formulierung. Ein typisches Beispiel ist die Bildung der Summe einer Sequenz von Zahlen (diesmal in musterbasierter Notation):

$$\begin{array}{ll}\text{DEF } sum\ \Diamond & = 0\\ \text{DEF } sum(x \cdot\colon rest) & = x + sum(rest) \quad -- \textit{ lineare Rekursion}\end{array}$$

Hier finden nach dem rekursiven Aufruf noch weitere Berechnungen statt. Man sagt auch, dass die Operation $(x + \_)$ „nachklappert“. Diese Form wird als ***lineare Rekursion*** bezeichnet. Man kann das optisch noch deutlicher hervorheben, indem man den Rumpf mit Hilfe von LET- oder WHERE-Deklarationen strukturiert. Dann haben linear rekursive Funktionen folgendes Muster:

$$\begin{array}{l}\text{DEF } f(x) = z \text{ WHERE } a = pre(x)\\ \qquad\qquad\qquad\qquad\quad r = f(a) \qquad\quad -- \textit{ lineare Rekursion}\\ \qquad\qquad\qquad\qquad\quad z = post(x, r)\end{array}$$

Im obigen *sum*-Beispiel entspricht $z = post(x, r)$ der Operation $z = x + r$.

Weil das Argument x in der nachklappernden Operation *post* noch gebraucht wird, muss der Compiler einen Stack vorsehen, in dem dieses Argument zwischengespeichert wird. Dieser Stack macht Aufwand und führt daher zu einem Effizienzverlust.

Deshalb wird es zu einer interessanten Frage, wie man linear rekursive Funktionen in Tail-rekursive Funktionen umwandeln kann.

Eine Standardtechnik besteht in einer ***Einbettung***: Man definiert – orientiert an der Form der gegebenen Funktion f – eine neue Funktion $\widetilde{f}$, die in geeignetem Sinne eine „Verallgemeinerung“ von f darstellt. Dann versucht man, durch ein bisschen Rechnen diese neue Funktion in Tail-rekursive Form

[9] Dieses deutsch-englische Mischwort ist zwar hässlich, hat sich aber in der Literatur eingebürgert, weshalb wir die Bezeichnung hier beibehalten.

zu bringen. Bevor wir diese Methodik weiter erläutern, illustrieren wir sie an dem konkreten Beispiel der obigen Funktion *sum*.

Wir führen für den nachklappernden Teil einen weiteren Parameter z ein und definieren folgende Einbettung:

$$\text{DEF } \widetilde{sum}(seq, z) = z + sum(seq)$$

Weil die Addition das neutrale Element 0 besitzt, ist *sum* als Spezialfall von $\widetilde{sum}$ darstellbar (was den Begriff „Verallgemeinerung“ rechtfertigt):

$$sum(seq) = 0 + sum(seq) = \widetilde{sum}(seq, 0)$$

Der wesentliche Teil der Entwicklung besteht in dem Versuch, $\widetilde{sum}$ in eine Tail-rekursive Form zu bringen. Dazu betrachten wir die gleichen musterbasierten Fälle wie in der Originalfunktion *sum* und wenden ein bisschen Mathematik an:

$$\begin{array}{lll} \widetilde{sum}(\lozenge, z) & = z + sum(\lozenge) & -- \textit{ Definition von } \widetilde{sum} \\ & = z + 0 & -- \textit{ Definition von sum} \\ & = z & -- \textit{ Mathematik} \end{array}$$

$$\begin{array}{lll} \widetilde{sum}(x \cdot\!: rest, z) & = z + sum(x \cdot\!: rest) & -- \textit{ Definition von } \widetilde{sum} \\ & = z + (x + sum(rest)) & -- \textit{ Definition von sum} \\ & = (z + x) + sum(rest) & -- \textit{ Mathematik} \\ & = \widetilde{sum}(rest, z + x) & -- \textit{ Definition von } \widetilde{sum} \end{array}$$

Das Ergebnis dieser Berechnungen lässt sich in ein neues Definitionssystem umwandeln. Dieses System ist jetzt Tail-rekursiv!

$$\begin{array}{lll} \text{DEF } sum(seq) & = \widetilde{sum}(seq, 0) & -- \textit{ Einbettung} \\ \text{DEF } \widetilde{sum}(\lozenge, z) & = z & \\ \text{DEF } \widetilde{sum}(x \cdot\!: rest, z) & = \widetilde{sum}(rest, z + x) & -- \textit{ Tail-Rekursion} \end{array}$$

Bei dieser Rechnung haben wir zwei Eigenschaften der Addition gebraucht: sie ist assoziativ und hat das neutrale Element 0. Deshalb können wir die Essenz dieser Berechnung in eine allgemeine Regel fassen.

Regel (Tail-Rekursion modulo Assoziativität)

Eine Funktion, die folgendem Schema genügt (lineare Rekursion)

$$\begin{array}{ll} \text{DEF } \mathcal{F}(\mathcal{A}) & = \mathcal{T} \\ \text{DEF } \mathcal{F}(\mathcal{B}) & = \mathcal{R} \oplus \mathcal{F}(\mathcal{P}) \quad -- \textit{ lineare Rekursion} \end{array}$$

kann in ein Tail-rekursives Schema transformiert werden:

$$\begin{array}{ll} \text{DEF } \mathcal{F}(\mathcal{X}) & = \widetilde{\mathcal{F}}(\mathcal{X}, \mathcal{E}) \qquad -- \textit{ Einbettung (neutrales Element)} \\ \text{DEF } \widetilde{\mathcal{F}}(\mathcal{A}, \mathcal{Z}) & = \mathcal{Z} \oplus \mathcal{T} \\ \text{DEF } \widetilde{\mathcal{F}}(\mathcal{B}, \mathcal{Z}) & = \widetilde{\mathcal{F}}(\mathcal{P}, \mathcal{Z} \oplus \mathcal{R}) \quad -- \textit{ Tail-Rekursion} \end{array}$$

Voraussetzung dafür ist, dass der Operator „ $\oplus$ “ assoziativ ist und ein neutrales Element „$\mathcal{E}$“ besitzt.

In dieser Regel stehen $\mathcal{A}$ und $\mathcal{B}$ für Konstruktorterme, $\mathcal{T}$, $\mathcal{P}$ und $\mathcal{R}$ für Ausdrücke in den Variablen dieser Muster, sowie $\mathcal{F}$, $\widetilde{\mathcal{F}}$, $\mathcal{X}$ und $\mathcal{Z}$ für Identifier.

Übrigens: das neutrale Element ist nicht unbedingt nötig; es geht auch mit Assoziativität alleine, wenn man eine etwas aufwendigere Einbettung in Kauf nimmt.

$$\begin{array}{ll} \text{DEF } \mathcal{F}(\mathcal{A}) & = \mathcal{T} \qquad\qquad -- \textit{ sofortige Terminierung} \\ \text{DEF } \mathcal{F}(\mathcal{B}) & = \widetilde{\mathcal{F}}(\mathcal{P},\mathcal{R}) \quad -- \textit{ Einbettung} \end{array}$$

Die Definition der Funktion $\widetilde{\mathcal{F}}$ bleibt unverändert.

Anmerkung: Neben der Assoziativität der nachklappernden Operation gibt es noch weitere algebraische Eigenschaften, mit deren Hilfe sich lineare Rekursion in Tail-Rekursion umwandeln lässt (Details findet man z. B. in [18]). In Kapitel 13 werden wir eine besonders wichtige Variante kennen lernen.

1.4.2 Ein „universeller Trick“: Continuations

Wenn Assoziativität nicht gegeben ist und auch keine der anderen Transformationen anwendbar ist, dann bleibt noch ein Trick übrig, der – zumindest formal – immer klappt. Man verwendet eine so genannte ***Continuation***, also eine Funktion, die – intuitiv gesprochen – die „Fortsetzung“ der Berechnung repräsentiert.

Warnung: Wir weisen schon jetzt darauf hin, dass dieser „universelle“ Trick im Allgemeinen ein bisschen Augenwischerei ist. (Genaueres am Ende dieses Abschnitts.)

Beispiel 1.2 (Anwendung von Continuations)

Wir betrachten die Auswertung eines Polynoms

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{1.1}$$

Diese Auswertung geschieht normalerweise nach dem so genannten *Hornerschema*:

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (\cdots + x \cdot (a_{n-1} + x \cdot (a_n)) \ldots))) \tag{1.2}$$

Das lässt sich sofort in ein (linear rekursives) Programm umschreiben, bei dem wir die Koeffizienten $a_0, \ldots, a_n$ als Sequenz repräsentieren.

$$\begin{array}{ll} \text{FUN } \mathit{horner}\colon & \mathit{Seq\ Real} \times \mathit{Real} \to \mathit{Real} \\ \text{DEF } \mathit{horner}(\Diamond,\ x) & = 0 \\ \text{DEF } \mathit{horner}(a\ \cdot\!\!:\mathit{rest},\ x) & = a + x \cdot \mathit{horner}(\mathit{rest}, x) \end{array}$$

Da die nachklappernde Operation $(a + x \cdot \_\ )$ nicht assoziativ ist, kann man die schematische Regel nicht anwenden. (Aber man kann versuchen, mit

einer geschickten Einbettung zum Erfolg zu gelangen. Wir überlassen das dem interessierten Leser.) Wir benutzen dieses Beispiel, um den universellen Trick mit den Continuations zu illustrieren.

Wir nehmen eine Einbettung vor, die als zusätzliches Argument die Continuation Γ enthält:

$$\begin{array}{l}\text{FUN } h\colon \mathit{Seq}\ \mathit{Real} \times \mathit{Real} \times (\mathit{Real} \to \mathit{Real}) \to \mathit{Real}\\ \text{DEF } h(s,\ x,\ \Gamma) = \Gamma(\mathit{horner}(s,\ x))\end{array}$$

Jetzt treiben wir unser übliches mathematisches Spiel:

$$\begin{array}{ll} h(\Diamond,\ x,\ \Gamma) & = \Gamma(\mathit{horner}(\Diamond,\ x))\\ & = \Gamma(0)\\ h(a\ \cdot\!\!:\mathit{rest}, x, \Gamma) & = \Gamma(\mathit{horner}(a\ \cdot\!\!:\mathit{rest},\ x))\\ & = \Gamma(a + x \cdot \mathit{horner}(\mathit{rest}, x))\\ & = \Gamma \circ (a + x \cdot \_)\ (\mathit{horner}(\mathit{rest}, x))\\ & = h(\mathit{rest},\ x,\ \Gamma \circ (a + x \cdot \_)) \end{array}$$

Damit erhalten wir das neue Definitionssystem:

$$\begin{array}{ll}\text{DEF } \mathit{horner}(s, x) & = h(s, x, \mathit{id})\\ \text{DEF } h(\Diamond, x, \Gamma) & = \Gamma(0)\\ \text{DEF } h(a\ \cdot\!\!:\mathit{rest}, x, \Gamma) & = h(\mathit{rest},\ x,\ \Gamma \circ (a + x \cdot \_))\end{array}$$

Warum klappt das immer? Der Grund ist ganz einfach: Die Funktionskomposition ist assoziativ und hat die Identitätsfunktion als neutrales Element. *Aber da ist ein Wermutstropfen*: Zwar kann man mit Hilfe von Continuations jede linear rekursive Funktion formal in eine Tail-rekusive Funktion verwandeln, aber damit ist im Allgemeinen kein Effizienzgewinn verbunden! Intern ist immer noch die gleiche Arbeit zu verrichten. (Compilertechnisch gesehen wird der Stack in den Heap verlagert, was den Aufwand sogar erhöht.)

Im obigen Beispiel der Funktion *horner* wird eine lange Continuation aufgebaut

$$\mathit{id} \circ (a_0 + x \cdot \_) \circ (a_1 + x \cdot \_) \circ \ldots \circ (a_n + x \cdot \_),$$

die am Schluss auf den Wert 0 angewandt wird. Der Aufbau dieser langen Funktionskomposition kostet intern natürlich sehr viel Zeit und Speicherplatz, so dass der Gewinn durch die Tail-Rekursion mehr als aufgebraucht wird.

Beispiel 1.3 (Exkurs: Spielen mit Continuations)

Welch skurrile Spielereien man mit diesen Continuations treiben kann, zeigt besonders deutlich folgendes kleine Beispiel. (Dieses Beispiel geht auf Oege de Moor zurück; wir präsentieren es hier in adaptierter Form.) Gegeben sei ein Baum mit natürlichen Zahlen an den Knoten. Er soll so umgeformt werden, dass an allen Knoten der maximale Wert steht, der im Originalbaum vorkommt. Man erwartet, dass man dazu zwei Baumdurchläufe braucht: einen, um das Maximum zu suchen, und einen zweiten, um den neuen Baum zu bau-

en. Aber es gibt ein Programm, das dieses Problem – zumindest scheinbar – in einem Durchlauf löst. Wir arbeiten uns in zwei Schritten zu diesem Programm vor.

Zunächst definieren wir den Baumtyp im Stil von HASKELL mit Hilfe von Currying:

$$\begin{array}{lll} \text{TYPE } \mathit{Tree} & = & \mathit{tree}\ \mathit{Tree}\ \mathit{Nat}\ \mathit{Tree} \quad -\!-\ \mathit{innerer\ Knoten} \\ & \mid & \mathit{leaf}\ \mathit{Nat} \qquad\qquad\qquad\quad -\!-\ \mathit{Blatt} \end{array}$$

Über diesem Typ definieren wir dann eine Funktion *convert*, die zu einem gegebenen Baum ein Paar von Werten liefert, bestehend aus dem maximalen Knoten des Baumes und einer Funktion, die aus diesem maximalen Wert den gewünschten neuen Baum erzeugt.

$$\begin{array}{l} \text{FUN } \mathit{convert}\colon \mathit{Tree} \to \mathit{Nat} \times (\mathit{Nat} \to \mathit{Tree}) \\ \text{DEF } \mathit{convert}(\mathit{leaf}\ x) = (x,\ \lambda m \bullet \mathit{leaf}\ m) \\ \text{DEF } \mathit{convert}(\mathit{tree}\ \mathit{left}\ x\ \mathit{right}) = \text{LET } (m_1, \varphi_l) = \mathit{convert}(\mathit{left}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (m_2, \varphi_r) = \mathit{convert}(\mathit{right}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad m = \mathit{max}(x, m_1, m_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{IN} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (m,\ \lambda m \bullet \mathit{tree}\ \varphi_l(m)\ m\ \varphi_r(m)) \end{array}$$

Der Aufruf dieser Funktion für einen gegebenen Baum t erfolgt dann in der Form

$$\ldots\ \text{LET } (m, \varphi) = \mathit{convert}(t)\ \text{IN } \varphi(m)\ \ldots$$

Als zweiten Schritt macht man im Wesentlichen das Ergebnis zu einem weiteren Argument, wobei man allerdings die Funktionalität des Parameters φ noch adaptieren muss:

$$\begin{array}{l} \text{FUN } \mathit{conv}\colon \mathit{Tree} \to (\mathit{Nat} \times (\mathit{Nat} \to \mathit{Tree} \to \mathit{Tree})) \to \mathit{Nat} \times (\mathit{Nat} \to \mathit{Tree}) \\ \text{DEF } \mathit{conv}(\mathit{leaf}\ x)(m, \varphi) = (\mathit{max}(x, m), \lambda m \bullet \varphi\ m\ (\mathit{leaf}\ m)) \\ \text{DEF } \mathit{conv}(\mathit{tree}\ \mathit{left}\ x\ \mathit{right})(m, \varphi) = \\ \quad \text{LET } m_0 = \mathit{max}(m, x) \\ \qquad (m_1, \varphi_l) = \mathit{conv}(\mathit{left})(m_0, \varphi) \\ \qquad (m_2, \varphi_r) = \mathit{conv}(\mathit{right})(m_1, \lambda m \bullet \lambda t \bullet \mathit{tree}\ \varphi_l(m)\ m\ t) \\ \quad \text{IN} \\ \qquad (m_2,\ \lambda m \bullet \varphi_r\ m) \end{array}$$

Der zweite Fall lässt sich etwas kompakter schreiben:

$$\begin{array}{l} \text{DEF } \mathit{conv}(\mathit{tree}\ \mathit{left}\ x\ \mathit{right})(m, \varphi) = \\ \quad \text{LET } (m_1, \varphi_l) = \mathit{conv}(\mathit{left})(\mathit{max}(m, x), \varphi) \\ \quad \text{IN} \\ \qquad \mathit{conv}(\mathit{right})(m_1, \lambda m \bullet \mathit{tree}\ \varphi_l(m)\ m) \end{array}$$

Der initiale Aufruf dieser Funktion muss als Argument im Wesentlichen Null und die Identität übergeben:

$$\ldots\ \text{LET } (m, \varphi) = \mathit{conv}(t)(0, \lambda m \bullet \lambda t \bullet t)\ \text{IN } \varphi(m)\ \ldots$$

Aber natürlich ist das ein Taschenspielertrick. Denn das zweite Ergebnis φ ist eine riesige Funktionskomposition, die alle Konstruktoren des Originalbaums enthält. Bei der Anwendung $\varphi(m)$ dieser Funktion auf das Maximum m wird deshalb die gesamte Struktur des Originalbaums rekonstruiert – und das ist natürlich der zweite Durchlauf.

Diese kleinen Beispiele zeigen, dass die Verwendung von Continuations mit Vorsicht erfolgen muss. Manchmal scheinen sie Probleme zu lösen, aber bei genauerem Hinsehen ist das nur ein oberflächliches Vortäuschen einer Lösung. Deshalb ist die Einführung von Continuations vor allem als Zwischenschritt zur Vorbereitung weiterer Transformationen interessant.[10]

1.4.3 Vereinfachung komplexerer Rekursionen

Lineare Rekursion ist immer noch eine recht einfache Rekursionsform. Richtig interessant wird es, wenn man komplexere Arten von Rekursion betrachtet. Ein berühmtes Beispiel ist die Fibonacci-Funktion:

$$\begin{array}{ll} \mathit{fib}(0) & = 1 \\ \mathit{fib}(1) & = 1 \\ \mathit{fib}(n+1) & = \mathit{fib}(n) + \mathit{fib}(n-1) \quad -- \textit{ Baum-Rekursion} \end{array}$$

Bei dieser Form spricht man von ***baumartiger Rekursion*** (kurz: ***Baum-Rekursion***), weil die Aufrufe baumartig verzweigen. Die Konsequenz ist in der Praxis dramatisch: Die Ausführung hat *exponentiellen Aufwand*.[11]

Die Addition ist assoziativ und hat das neutrale Element 0. Deshalb könnten wir versuchen, die entsprechende Transformation anzuwenden. Das Ergebnis hätte folgende Form:

$$\begin{array}{ll} \mathit{fib}(n) & = f(n, 0) \\ f(0, z) & = z + 1 \\ f(1, z) & = z + 1 \\ f(n+1, z) & = f(n,\ z + f(n-1, 0)) \quad -- \textit{ geschachtelte Rekursion} \end{array}$$

Dies ist eine weitere komplexe Rekursionsform, die so genannte ***geschachtelte Rekursion***, bei der ein rekursiver Aufruf als Argument einen weiteren rekursiven Aufruf enthält. Diese Form ist unter Effizienzgesichtspunkten genauso ungünstig wie die baumartige Rekursion. Deshalb führt dieser naive Versuch in eine Sackgasse.

---

[10] Eine ganz andere – und sehr wichtige – Verwendung von Continuations werden wir in Kapitel 17 kennen lernen: Sie sind ein wesentlicher Bestandteil der so genannten Monaden-basierten Ein-/Ausgabe.

[11] Man kann sich den Spass machen, diese Funktion in irgendeiner beliebigen Programmiersprache aufzuschreiben und dann der Reihe nach $\mathit{fib}(10)$, $\mathit{fib}(20)$, $\mathit{fib}(30)$, $\mathit{fib}(40)$, ... aufzurufen. Dabei lernt man überraschend schnell, was „exponentieller Aufwand“ bedeutet. (Um das Experiment spannender zu machen, kann man vorher schätzen, ab wann das Ganze nicht mehr machbar ist.)

Trotzdem können wir Funktionen wie *fib* unter geeigneten Randbedingungen in einfachere Form verwandeln. Das soll im Folgenden gezeigt werden.

Allerdings nehmen wir dazu nicht *fib*, sondern eine andere Funktion, die noch etwas kniffliger aussieht. Aber die Entwicklung lässt sich völlig analog auf *fib* übertragen (was wir dem interessierten Leser als Übung überlassen).

Wir hatten weiter vorne schon die etwas artifizielle Funktion *fusc* kennen gelernt, die hier interessant ist, weil sie ein recht komplexes Rekursionsmuster aufweist: In einem Zweig hat sie Tail-Rekursion, im anderen Baum-Rekursion.

$$
\begin{array}{lll}
\text{DEF } \mathit{fusc}(0) & = 1 & \\
\text{DEF } \mathit{fusc}(1) & = 1 & \\
\text{DEF } \mathit{fusc}(2n) & = \mathit{fusc}(n) & -- \textit{ Tail-Rekursion} \\
\text{DEF } \mathit{fusc}(2n+1) & = \mathit{fusc}(n) + \mathit{fusc}(n+1) & -- \textit{ Baum-Rekusion}
\end{array}
$$

Wir wollen zeigen, dass sich auch solche uneinheitlichen Formen sytematisch behandeln lassen. Wir folgen hier einer besonders eleganten Darstellung, die von David Gries präsentiert wurde. Anstatt wie üblich mehrere Hilfsparameter einzuführen, verwenden wir die Eleganz der mathematischen Vektor- und Matrizenrechnung. (Letztlich ist ein Vektor v nichts anderes als eine flexible Kurznotation für mehrere Variablen $v_1, \ldots, v_n$.)

Indem wir die beiden rekursiven Aufrufe in einen Vektor verwandeln, können wir den letzten Rekursionszweig als Skalarprodukt schreiben. Und mit dem kleinen Trick, dass Multiplikation mit 0 einen Wert annulliert, können wir auch den anderen Rekursionszweig in die gleiche Form bringen.[12]

$$
\begin{array}{lll}
\text{DEF } \mathit{fusc}(0) & = 1 & \\
\text{DEF } \mathit{fusc}(1) & = 1 & \\
\text{DEF } \mathit{fusc}(2n) & = (1 \;\; 0) \cdot \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n+1) \end{pmatrix} & -- \textit{ Baum-Rekursion} \\
\text{DEF } \mathit{fusc}(2n+1) & = (1 \;\; 1) \cdot \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n+1) \end{pmatrix} & -- \textit{ Baum-Rekursion}
\end{array}
$$

Jetzt führen wir eine Einbettung ein, indem wir (auf Verdacht) eine Hilfsfunktion f definieren, die gerade der Vektorbildung entspricht.

$$\text{DEF } f(n) = \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n+1) \end{pmatrix}$$

Diese neue Funktion f stützt sich nach wie vor auf *fusc* ab. Deshalb versuchen wir als nächstes, direkte Rekursionsgleichungen für f abzuleiten, die dem Muster der Definition von *fusc* folgen. Für die Terminierungszweige ist das sehr einfach

[12] *Vorsicht*! Diese Definition ist nicht ganz korrekt. Unter Call-by-value würde *fusc*(2) nicht terminieren. Denn die Tatsache, dass der Wert durch die Multiplikation mit 0 annulliert wird, hilft unter dieser Auswertungsstrategie nichts. Aber weil es sich nur um eine Zwischenform handelt, sind wir etwas großzügiger und ignorieren dieses vorübergehende Phänomen.

$$f(0) = \begin{pmatrix} \mathit{fusc}(0) \\ \mathit{fusc}(1) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
$$f(1) = \begin{pmatrix} \mathit{fusc}(1) \\ \mathit{fusc}(2) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Bei den beiden Rekursionszweigen müssen wir etwas mehr rechnen. Wir beginnen mit dem ersten Zweig $f(2n)$:

$$\begin{aligned} f(2n) &= \begin{pmatrix} \mathit{fusc}(2n) \\ \mathit{fusc}(2n+1) \end{pmatrix} && -- \textit{ Definition von } f \\ &= \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n) + \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Definition von fusc} \\ &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Mathematik} \\ &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ Definition von } f \end{aligned}$$

Der andere Zweig $f(2n+1)$ lässt sich ganz analog durchrechen:

$$\begin{aligned} f(2n+1) &= \begin{pmatrix} \mathit{fusc}(2n+1) \\ \mathit{fusc}(2n+2) \end{pmatrix} && -- \textit{ Definition von } f \\ &= \begin{pmatrix} \mathit{fusc}(n) + \mathit{fusc}(n+1) \\ \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Definition von fusc} \\ &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Mathematik} \\ &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ Definition von } f \end{aligned}$$

Diese Gleichungen benutzen wir jetzt als Definitionssystem. Dazu müssen wir uns prinzipiell noch davon überzeugen, dass die so eingeführten rekursiven Definitionen terminieren (hier trivial) und dass alle Operationen wohl definiert sind. Der Wert von $\mathit{fusc}(n)$ ist aufgrund der Einbettung gerade die erste Komponente von $f(n)$.

$$\begin{aligned} \text{DEF } \mathit{fusc}(n) &= f(n).1 \\ \text{DEF } f(0) &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \text{DEF } f(1) &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \text{DEF } f(2n) &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ lineare Rekursion} \\ \text{DEF } f(2n+1) &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ lineare Rekursion} \end{aligned}$$

Da die Matrixmultiplikation assoziativ ist und ein neutrales Element hat, könnten wir die entsprechende Transformationregel anwenden. Allerdings würden wir dadurch 2×2-Matrizen als zusätzliche Argumente erhalten, was unnötig aufwendig ist, weil wir uns ja nur für den Wert $f(n).1$, also die erste Komponente des Resultatvektors, interessieren. Deshalb nehmen wir keine Einbettung mit Matrizen vor, sondern nur eine Einbettung mit Vektoren:

$$\text{DEF } \widetilde{f}(n,(z_1,z_2)) \;= (z_1 \;\; z_2) \cdot f(n)$$

Es gibt zwar keinen Vektor $(z_1 \;\; z_2)$, für den diese Multiplikation den Wert $f(n)$ unverändert liefert (dazu ist die Einheits*matrix* notwendig, nicht ein Vektor), aber da uns ja ohnehin nur die erste Komponente interessiert, genügt uns die Einbettung

$$\text{DEF } \mathit{fusc}(n) \;= f(n).1 \;= (1 \;\; 0) \cdot f(n)$$

Für $\widetilde{f}(n)$ können wir unsere Standardrechnung durchführen (was wir dem interessierten Leser als Übung überlassen), so dass am Ende das folgende System entsteht:

$$\begin{array}{ll}
\text{DEF } \mathit{fusc}(n) & = \widetilde{f}(n,(1,0)) \\
\text{DEF } \widetilde{f}(0,(z_1,z_2)) & = z_1 + z_2 \\
\text{DEF } \widetilde{f}(1,(z_1,z_2)) & = z_1 + z_2 \\
\text{DEF } \widetilde{f}(2n,(z_1,z_2)) & = \widetilde{f}(n,\,(z_1+z_2,\,z_2)) \\
\text{DEF } \widetilde{f}(2n+1,(z_1,z_2)) & = \widetilde{f}(n,\,(z_1,\,z_1+z_2))
\end{array}$$

Natürlich hätten wir diese weitere Einbettung auch gleich bei der ersten Hälfte der Entwicklung mit einbauen können, so dass wir direkt von der Baum-Rekursion zur Tail-Rekursion gekommen wären, ohne die lineare Rekursion zwischenzuschalten. Aber das hätte die Präsentation wesentlich unleserlicher gemacht.

Diese kleinen Beispiele sollten hinreichend illustrieren, wie gut man mit funktionalen Programmen arbeiten kann. Dabei ist es prinzipiell egal, ob diese Umformungen „von Hand“ erfolgen oder automatisch im Compiler.

1.5 OPAL, ML und HASKELL

Die meisten funktionalen Sprachen sind sich relativ ähnlich und lassen im Allgemeinen alle hier diskutierten Formen der Notation zu. Meist wird die gleichungsartige Definition von Funktionen gegenüber der λ-Notation als Normalfall vorgezogen. Natürlich gibt es aber syntaktische Unterschiede bzgl. der Verwendung von Symbolen und Schlüsselwörtern.

Beispiel: Wie man in OPAL ein Filter-Funktional definiert und auf eine anonyme Funktion und eine Liste anwendet, haben wir schon gesehen. In HASKELL und ML sieht das bis auf kleine syntaktische Unterschiede im Prinzip genauso aus. In HASKELL schreibt man den Typ der Sequenzen von Elementen eines Typs a als $[a]$ und unsere Operation „ $\cdot$: “ einfach als „: “. Die

Typisierung wird mit „ :: “ notiert. Damit sieht *filter* dann folgendermaßen aus:

```
filter :: ( a → Bool ) → [a] → [a]                    -- HASKELL-Notation
filter p [] = []
filter p (x: xs) = if p x then x: (filter p xs) else filter p xs
```

Eine Anwendung hat dann z. B. folgende Form:

```
filter(\ x → x > 0) [3, 5, −2, 0, 18, −12]            -- HASKELL-Notation
```

In ML sieht das Ganze folgendermaßen aus:

```
fun filter p [] = [] |                                 -- ML-Notation
    filter p (x :: xs) = if p x then x :: (filter p xs) else filter p xs
```

Eine Anwendung hat dann z. B. folgende Form:

```
filter (fn x => x > 0) [3, 5, −2, 0, 18, −12]          -- ML-Notation
```

Bestimmte Standardfunktionen und -typen, die sich wie in den meisten Programmiersprachen in vordefinierten Strukturen (OPAL, ML) oder Modulen (HASKELL) befinden, können unterschiedliche Namen und Varianten haben, auch wenn sich in vielen Fällen hierfür feste Bezeichner eingebürgert haben. Beispielsweise wird das Reduce-Funktional $\not$ in HASKELL und ML als *foldr* bezeichnet; die Reduce-Version für die inverse Richtung $\not$ heißt dort *foldl*. Der Datentyp *option* aus der OPAL-Struktur *Option* ermöglicht es, mit optionalen Werten umzugehen und damit mit der Situation, dass eine Funktionsauswertung auch fehlschlagen kann. HASKELL und ML haben entsprechende Datentypen *Maybe* bzw. *option* mit gleicher Wirkung.

Während man in OPAL bei der Deklaration einer Funktion den Typ explizit angeben muss und das System prüft, ob der Rumpf typkorrekt ist, können in HASKELL und ML Typangaben (in den meisten Fällen) weggelassen werden, da das System diese inferiert.

Wie schon oben erwähnt, können in OPAL Strukturen polymorph sein, d. h. polymorphe Datentypen und Funktionen enthalten. Diese kann man dann im Programm mit geeigneten Datentypen instanziieren. Im Gegensatz dazu wird in ML und HASKELL Polymorphie nicht global für ganze Strukturen festgelegt, sondern individuell für jeden Typ und jede Funktion einzeln.

Auf einige dieser Unterschiede werden wir in späteren Kapiteln noch genauer eingehen.

2

Faulheit währt unendlich

Faulheit ist die Furcht vor bevorstehender Arbeit.
Cicero

Blinder Eifer schadet nur.
Sprichwort

In Abschnitt 1.3 haben wir funktionale Sprachen entsprechend ihrer Auswertungsstrategien in *eager* (bzw. *strikte*) und *lazy* (oder *nichtstrikte*) Sprachen unterschieden. Strikte Sprachen werten alle Argumente vollständig aus, bevor sie eine Funktion applizieren, während lazy Sprachen ihre Argumente nur bei Bedarf auswerten. Dadurch können in strikten Sprachen nur Funktionen rekursiv definiert werden, Datenstrukturen hingegen nicht, denn die strikte Auswertung einer rekursiv definierten Datenstruktur würde nicht terminieren. Im Gegensatz dazu lassen lazy Sprachen die Definition und Verwendung unendlicher Datenstrukturen zu. Man aber auch in strikten Sprachen unendliche Datenstrukturen durch Funktionen *simulieren*, die ein solches unendliches Objekt schrittweise expandieren. Diese Simulation kann allerdings den wichtigen Aspekt des *Sharings* (vgl. Abschnitt 1.3.2) nicht nachbilden.

In diesem Kapitel werden wir die Vorteile der Verwendung unendlicher Datenstrukturen demonstrieren und zeigen, wie man auch in strikten Sprachen, d. h. in Sprachen mit einer eager Auswertung, mit unendlichen Objekten umgehen kann.

2.1 Unendliche Objekte: die Idee

Während bei Funktionen rekursive Definitionen gang und gäbe sind, ist das bei Objekten in strikten Sprachen wie OPAL oder ML nicht der Fall. Hier sind Gleichungen wie die folgenden verboten:

```
FUN a b: List Nat   -- (unendliche) Listen natürlicher Zahlen
DEF a = 1 ·: a      -- in OPAL oder ML so nicht möglich
DEF b = 1 ·: 2 ·: b -- in OPAL oder ML so nicht möglich
```

Und das mit gutem Grund. Denn strikte Sprachen werten ihre Argumente vollständig aus, bevor sie eine Funktion applizieren. Würden wir also die Sequenzen a oder b als Argumente in einer Funktion verwenden, so würde unsere Berechnung nicht terminieren. Dabei haben wir eigentlich eine ganz gute Intuition für das, was diese beiden Gleichungen bewirken sollten:

$a = \langle 1, 1, 1, 1, 1, \ldots \rangle$ $\quad --$ *unendliche Liste*
$b = \langle 1, 2, 1, 2, 1, \ldots \rangle$ $\quad --$ *unendliche Liste*

Wenn wir annehmen, dass auch alle Operatoren wie *Map*, *Filter*, *Reduce* etc. für unendlichen Sequenzen funktionieren, können wir sogar schreiben:

DEF $powers = x \cdot\!: (f * powers)$

Das ist nichts anderes als die unendliche Liste

$powers = \langle x, f\,x, f^2\,x, f^3\,x, f^4\,x, \ldots \rangle$

Das sieht man sehr gut, wenn man die Folgen geeignet übereinander legt:

$$\begin{array}{lllllll} x & \cdot\!: & f\,x & \cdot\!: & f^2\,x & \cdot\!: & f^3\,x \;\; \cdot\!: \;\; \ldots \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ x \cdot\!: & f\,x & \cdot\!: & f^2\,x & \cdot\!: & f^3\,x & \cdot\!: \;\; \ldots \end{array} \quad \begin{array}{l} \widehat{=}\ powers \\ \widehat{=}\ x \cdot\!: (f * powers) \end{array}$$

Dass man mit so etwas schön arbeiten kann, zeigt Beispiel 2.1.

Beispiel 2.1 (Präfixsummen)

Wir wollen die so genannten Präfixsummen der Folge der natürlichen Zahlen ausrechnen, d. h. die Folge

$\langle (0), (0+1), (0+1+2), (0+1+2+3), (0+1+2+3+4), \ldots \rangle$

Zunächst erzeugen wir – analog zu *powers* oben – die unendliche Sequenz $\langle 1, 2, 3, 4, 5, \ldots \rangle$ aller natürlichen Zahlen.

DEF $allNats = 1 \cdot\!: (+1) * allNats$

Darauf wenden wir dann den *Zip*-Operator geeignet an:

DEF $sums = 0 \cdot\!: (sums \overset{+}{\bigtriangledown} allNats)$

Das lässt sich ganz einfach veranschaulichen. Man legt die beiden Sequenzen übereinander und sieht sich ihre Summenbildung an:

`allNats:`	1	2	3	4	5	6	...
+ `sums:`	0	1	3	6	10	15	...
=	1	3	6	10	15		...

Übrigens: Wenn man die Funktion $inits(s)$ benutzt, die zu einer Sequenz s die Sequenz aller Anfangssequenzen liefert (also z. B. $inits(allNats) = \langle \Diamond, \langle 1 \rangle, \langle 1, 2 \rangle, \langle 1, 2, 3 \rangle, \ldots \rangle$), kann man die obige Sequenz auch so erhalten:

DEF $sums = (+/) * inits(allNats)$

Und um das Ganze noch amüsanter zu gestalten: Auch *inits*(*allNats*) kann direkt durch eine rekursive Gleichung dargestellt werden:

FUN *allNatInits*: *List* (*List Nat*)
DEF *allNatInits* = $\Diamond$ ·: (*allNatInits* $\dot{\curlyvee}$ *allNats*)

Aber damit wollen wir unseren Spieltrieb zügeln und uns den ernsteren Fragen des zugehörigen Sprachdesigns zuwenden.

2.2 LAZY, QUOTE und UNQUOTE

Im Folgenden wollen wir erörtern, wie sich diese Ideen in strikte Sprachen einbauen lassen. Dabei nehmen wir zunächst eine echte Spracherweiterung vor und diskutieren später, wie sich das wenigstens näherungsweise im strikten OPAL oder ML simulieren lässt. Das Grundprinzip besteht einfach darin, dass bei Funktionen für die nicht-strikten Parameter die Auswertung der Argumente unterdrückt wird. (Dieses Prinzip ähnelt dem „quote“ von LISP.)

FUN _ $\barwedge$ _: *Bool* × LAZY *Bool* → *Bool* −− *sequenzielles Und*
DEF *a* $\barwedge$ *b* = IF *a* THEN *b* ELSE *false* FI

FUN _ $\dot{\curlyvee}$ _: *Bool* × LAZY *Bool* → *Bool* −− *sequenzielles Oder*
DEF *a* $\dot{\curlyvee}$ *b* = IF *a* THEN *true* ELSE *b* FI

Ohne den Zusatz LAZY würde das nicht funktionieren: In einem Ausdruck wie

... IF $y \neq 0 \barwedge \frac{x}{y} > 1$ THEN ...

würden in einer strikten Sprache beide Ausdrücke, also $y \neq 0$ ebenso wie $\frac{x}{y} > 1$, erst einmal vollständig zu Booleschen Werten reduziert, bevor der Rumpf von $\barwedge$ ausgeführt würde. Damit wäre für den Fall $y = 0$ das Unglück aber schon passiert.

Anmerkung: Spezielle Operatoren für sequenzielles Und und Oder gibt es z. B. auch in Sprachen wie C *oder* JAVA*; dort werden sie als* `&&` *bzw.* `||` *geschrieben. Dabei handelt es sich aber um zwei singuläre, vordefinierte Sprachfeatures. Im Gegensatz dazu stellen wir einen generellen Mechanismus bereit, mit dem Programmierer beliebige Operatoren dieser Art selbst definieren können.*

Um den Effekt von LAZY-Parametern zu verdeutlichen, führen wir die beiden Schlüsselwörter QUOTE und UNQUOTE ein, die wir an den passenden Stellen vor die entsprechenden Ausdrücke setzen. Mit QUOTE wird der Ausdruck „geschützt“, d. h. seine Ausführung erst einmal verhindert.

... IF $y \neq 0 \barwedge$ QUOTE$(\frac{x}{y} > 1)$ THEN ...

In der Definition der Funktion $\barwedge$ muss für das Argument *b* die Ausführung erzwungen werden, sobald sein Wert gebraucht wird. Dazu dient UNQUOTE:

DEF *a* $\barwedge$ *b* = IF *a* THEN UNQUOTE *b* ELSE *false* FI

Dieses Zusammenspiel von QUOTE und UNQUOTE erklärt die Begriffsbildung *verzögerte Auswertung* (engl.: *lazy evaluation*). Allerdings wollen wir aus Gründen der Lesbarkeit verhindern, dass die Schlüsselwörter QUOTE und UNQUOTE dauernd geschrieben werden müssen. Erfreulicherweise brauchen wir dazu aber keinen speziellen Mechanismus zu erfinden, sondern können die bekannten Subtyp- und Casting-Prinzipien verwenden (s. Kapitel 7), die in modernen Compilern ohnehin vorhanden sind.

2.2.1 LAZY als generischer Typ

Wir betrachten LAZY als generischen Typ. Das heißt, wir betrachten die Konstruktion, als ob sie folgendermaßen definiert wäre:

TYPE LAZY α = QUOTE (UNQUOTE : α) −− *Vorsicht! Spezielles Feature!*

Dies ist eine *echte Spracherweiterung*, denn in einer strikten Sprache wäre eine Anwendung wie QUOTE($\frac{x}{y} > 1$) eben kein Schutz vor Undefiniertheit. Deshalb ist QUOTE keine normale Funktion, sondern ein spezielles Konstrukt (wie z. B. auch IF.THEN.ELSE.FI).

Wir gehen allerdings ab jetzt davon aus, dass – in Analogie zum Casting-Mechanismus bei Subtypen (s. Kapitel 7) – die Typanpassungen mittels QUOTE und UNQUOTE automatisch erzeugt werden, so dass wir in der Tat unsere Ausgangsnotation vom Anfang dieses Abschnitts beibehalten können.

2.2.2 Simulation in strikten Sprachen wie OPAL oder ML

Wenn man das obige Zusammenspiel von QUOTE und UNQUOTE genauer ansieht, erkennt man sofort, wie sich ihr Effekt in strikten Sprachen wie OPAL oder ML zumindest teilweise simulieren lässt: Man muss einfach nur λ-Abstraktionen schreiben: Bei der Deklaration sehen wir als Parameter eine nullstellige Funktion vor, die dann im Rumpf zu applizieren ist:

FUN _ ∧ _ : *Bool* × (() → *Bool*) → *Bool*
DEF a ∧ b = IF a THEN $b()$ ELSE *false* FI

Bei der Applikation von ∧ müssen wir ein entsprechendes „leeres λ-Lifting“ vornehmen, um die Typkorrektheit zu erhalten:

... IF $y \neq 0$ ∧ $(\lambda \bullet \frac{x}{y} > 1)$ THEN ...

Die Simulation erfolgt also nach folgenden systematischen Substitutionsregeln („*Cut-Copy-Paste*“):

- LAZY α ist durch eine nullstellige Funktion (() → α) zu ersetzen.
- QUOTE E ist durch das leere λ-Lifting ($\lambda \bullet E$) zu ersetzen.
- UNQUOTE E ist durch die Applikation $E()$ zu ersetzen.

Wie schon erwähnt, kann diese Simulation allerdings nicht den wichtigen Aspekt des Sharings (vgl. Abschnitt 1.3.2) nachbilden.

2.3 LAZY Listen

Wir wollen an einigen Beispielen studieren, wie sich mit Hilfe der LAZY-Konstruktion sehr elegante Programme schreiben lassen. Der wesentliche Punkt ist, dass wir die unendlichen Datenstrukturen vom Anfang dieses Kapitels jetzt tatsächlich erhalten. Wir verwenden folgenden Typ der ***unendlichen Listen*** (manchmal auch *lazy Listen* genannt):

```
TYPE List α = _ ·: _ (ft = α, rt = LAZY List α)   -- unendliche Listen
```

Genauer heißt das eigentlich Folgendes: Die Typdeklaration induziert die Konstruktoren und Selektoren (s. Kapitel 1 und 6):

```
FUN _ ·: _: α × LAZY List α → List α   -- Konstruktor
FUN ft: List α → α                     -- Selektor
FUN rt: List α → LAZY List α           -- Selektor
```

Insgesamt lassen sich drei Arten von Listen unterscheiden. Der Einfachheit halber verwenden wir in diesem Buch für alle drei den gleichen Namen *List* α. Welche Version gemeint ist, ergibt sich jeweils aus dem Kontext. In der Programmierung kann man diese Namensgleichheit ebenfalls realisieren, indem man die drei Versionen in getrennten Strukturen deklariert (s. Kapitel 4). Wir verwenden dabei den Hilfstyp *Empty*, der als einziges Element die leere Liste, geschrieben als „$\Diamond$“, enthält:

```
TYPE Empty = { ◊ }
```

- *Endliche Listen* werden ohne LAZY definiert, so dass der *Empty*-Fall zwingend notwendig ist.

```
TYPE List α = _ ·: _ (ft = α, rt = List α)   -- endliche Listen
            | Empty
```

- *Potenziell unendliche Listen* werden mit LAZY definiert, eröffnen aber durch die *Empty*-Variante auch die Chance zur endlichen Terminierung.

```
TYPE List α = _ ·: _ (ft = α, rt = LAZY List α)   -- pot. unendlich
            | Empty
```

- *Unendliche Listen* werden mit LAZY definiert und bieten keine Alternative zur endlichen Terminierung an.

```
TYPE List α = _ ·: _ (ft = α, rt = LAZY List α)   -- unendliche Listen
```

Die folgenden Überlegungen gelten prinzipiell für unendliche und potenziell unendliche Listen gleichermaßen, auch wenn wir in den Beispielen nur unendliche Listen verwenden.

Wir betrachten exemplarisch die *Map*-Funktion. Sie lässt sich folgendermaßen definieren:

```
FUN _ * : (α → β) → List α → List β
DEF f * (a ·: rest) = f a ·: (f * rest)
```

Wenn wir akkurater sind und die notwendigen Typanpassungen mittels QUOTE und UNQUOTE explizit hinschreiben, dann entspricht das folgender Definition (wobei wir jetzt, um der Klarheit willen, nicht mehr mit Patternmatching arbeiten):

$$\text{DEF } f * L = (f\ (ft\ L))\ \cdot\!:\ \text{QUOTE}\ (f * \text{UNQUOTE}\ (rt\ L))$$

Aus dieser vervollständigten Form sieht man, dass wegen des Schutzes durch QUOTE der Ausdruck $(f * \text{UNQUOTE}\ (rt\ L))$ zunächst *nicht* ausgeführt wird.

Was passiert, wenn wir jetzt einen Teilausdruck wie

$$\ldots\ ft(rt(f * L))\ \ldots$$

für irgendeine Liste $L = (a\ \cdot\!:\ \text{QUOTE}\ (b\ \cdot\!:\ \text{QUOTE}\ (c\ \cdot\!:\ \ldots)))$ ausrechnen wollen? Zunächst müssen wir natürlich wegen der Typkorrektheit noch ein UNQUOTE einfügen. Dann sehen wir durch Einsetzen sofort, dass folgender Ausdruck entsteht (wobei wir immer beachten müssen, dass bei Ausdrücken, die nicht durch QUOTE geschützt sind, die Argumente von innen nach außen auszuwerten sind):

$$\begin{aligned}
& ft(\text{UNQUOTE}\ (rt(\ f * L\ ))) \\
= \ & ft(\text{UNQUOTE}\ (rt(\ (f\ (ft\ L))\ \cdot\!:\ \text{QUOTE}\ (f * (\text{UNQUOTE}\ (rt\ L)))\ ))) \\
= \ & ft(\ \text{UNQUOTE}\ (\text{QUOTE}\ (f * (\text{UNQUOTE}\ (rt\ L))))\ ) \\
= \ & ft(\ f * (\text{UNQUOTE}\ (rt\ L))\ )
\end{aligned}$$

Jetzt müssen wir die tatsächliche Liste L betrachten, um weiterrechnen zu können:

$$\begin{aligned}
= \ & ft(\ f * (\text{UNQUOTE}\ (rt\ (a\ \cdot\!:\ \text{QUOTE}(b\ \cdot\!:\ \text{QUOTE}(c\ \cdot\!:\ \ldots))))\ ) \\
= \ & ft(\ f * (\text{UNQUOTE}\ (\text{QUOTE}(b\ \cdot\!:\ \text{QUOTE}(c\ \cdot\!:\ \ldots))))\ ) \\
= \ & ft(\ f * (b\ \cdot\!:\ \text{QUOTE}(c\ \cdot\!:\ \ldots))\ ) \\
= \ & ft(\ (f\ b)\ \cdot\!:\ \text{QUOTE}\ (f * (\text{UNQUOTE}\ (rt\ (b\ \cdot\!:\ \text{QUOTE}(c\ \cdot\!:\ \ldots))))\ ) \\
= \ & f\ b
\end{aligned}$$

Diese kleine Rechenübung zeigt zweierlei:

- Das Prinzip mit QUOTE und UNQUOTE funktioniert in der Tat. Ein Compiler kann also nach diesem Verfahren Code erzeugen.
- Lazy Konstrukte kosten Aufwand, weshalb Sprachen, die alles lazy erledigen, ineffizienter sind als strikte Sprachen. (Deshalb wird in solchen Sprachen oft versucht, mittels einer so genannten *Striktheitsanalyse* diejenigen Ausdrücke zu erkennen, die sich eager auswerten lassen.)

Deshalb arbeiten wir im Folgenden weiter mit strikten Sprachen, wobei wir aber das Schlüsselwort LAZY verwenden, um z. B. unendliche Datenstrukturen da verfügbar zu machen, wo sie hilfreich sind. Da wir außerdem QUOTE und UNQUOTE nie explizit hinschreiben (sondern sie vom Compiler generieren lassen), werden die entsprechenden Programme sehr elegant.

2.4 Programmieren mit LAZY Listen

Zur Abrundung unserer Diskussion von lazy Datenstrukturen wollen wir noch an einigen Beispielen vorführen, wie elegant sich gewisse Programmieraufgaben damit lösen lassen. Dabei betrachten wir drei Klassen von Programmieraufgaben:

- Erzeugung (potenziell) unendlicher Folgen,
- Approximationsaufgaben und
- Datenfluss („Ströme“; engl.: *streams*), insbesondere für Ein-/Ausgabe.

2.4.1 Unbeschränkte Folgen

In der Informatik gibt es eine Reihe von Aufgaben, bei denen man eine Folge von Werten errechnen soll, die *im Prinzip unbeschränkt* lang sein kann. Der Abbruch kann dann sehr flexibel gefordert sein, etwa „die ersten n Elemente“ oder „bis mindestens ein Element der Größe x erreicht ist“ etc. Wir betrachten zwei typische Vertreter dieser Klasse von Programmieraufgaben.

Beispiel 2.2 (Hammings Denksportaufgabe)

Von Hamming stammt eine hübsche kleine Aufgabe: Man bestimme die geordnete Menge (also aufsteigende Folge ohne Mehrfachvorkommen) aller Zahlen, die sich in die Primfaktoren 2, 3 und 5 zerlegen lassen, d. h. alle Zahlen der Form $2^i \cdot 3^j \cdot 5^k$ mit $i, j, k \geq 0$.

So trivial diese Aufgabe auf den ersten Blick aussieht, so verzwickt erweist sie sich im Detail – wenn man mit klassischen Variablen und Schleifen an die Sache herangeht. Wir wollen zeigen, wie trivial die Lösung wird, wenn man lazy Listen benutzt.

Sei also H die gewünschte Folge. Dann gilt offensichtlich, dass die drei Folgen *H2*, *H3* und *H5* Teilfolgen von H sind:

FUN *H H2 H3 H5*: *List Nat*
DEF $H2 = (2 \cdot) * H$
DEF $H3 = (3 \cdot) * H$
DEF $H5 = (5 \cdot) * H$

Umgekehrt sieht man auch sofort, dass (mit der offensichtlichen Funktion *merge* – s. unten) gilt:

DEF $H = 1 \cdot\!: merge(H2, H3, H5)$

Die 1 muss in H sein. Und wenn wir annehmen, dass alle Zahlen bis zu einem H_i in $H2 \cup H3 \cup H5$ enthalten sind, dann folgt sofort, dass auch H_{i+1} darin enthalten ist. Denn H_{i+1} entsteht aus einem H_j mit $j \leq i$ durch Multiplikation mit 2, 3 oder 5.

Die Funktion *merge* ist einfach:

```
FUN merge: List Nat × List Nat × List Nat → List Nat
DEF merge (a ·: A, b ·: B, c ·: C) =
      LET m = min(a, b, c)
      IN
      m ·: merge( IF m < a THEN a ·: A ELSE A FI,
                  IF m < b THEN b ·: B ELSE B FI,
                  IF m < c THEN c ·: C ELSE C FI )
```

Beispiel 2.3 (Primzahlen)

Ein Klassiker der Programmierung sind die Primzahlen nach dem Sieb des Eratosthenes. Mit lazy Listen geht auch das sehr schön. Die gesuchte Liste *primes* der Primzahlen erhält man als

```
FUN primes: List Nat
DEF primes = sieve (rt allNats)
```

Dabei ist *allNats* die schon früher beschriebene Liste aller natürlichen Zahlen, so dass *rt allNats* alle Zahlen ab der 2 umfasst. Die Funktion *sieve* ist ganz einfach, wobei wir den Test $p\nmid$ für „p teilt nicht ...“ verwenden:

```
FUN sieve: List Nat → List Nat
DEF sieve (p ·: rest) = p ·: sieve ( (p ∤) ◁ rest )
```

Um ihre Funktionsweise zu verstehen, sehen wir uns die ersten Schritte an, wobei wir mit „[“ die Stellen kennzeichnen, die mit QUOTE geschützt sind:

```
sieve ⟨2, [3, [4, [5, [6, [7, [8, [9, ...⟩ =
2 ·: sieve ((2 ∤) ◁ ⟨[3, [4, [5, [6, [7, [8, [9, ...⟩) =
2 ·: sieve (3 ·: [(2 ∤) ◁ ⟨4, [5, [6, [7, [8, [9, ...⟩) =
2 ·: 3 ·: sieve ((3 ∤) ◁ (2 ∤) ◁ ⟨4, [5, [6, [7, [8, [9, ...⟩) =
2 ·: 3 ·: sieve ((3 ∤) ◁ (2 ∤) ◁ ⟨5, [6, [7, [8, [9, ...⟩) =
2 ·: 3 ·: sieve ((3 ∤) ◁ (5 ·: [(2 ∤) ◁ ⟨6, [7, [8, [9, ...⟩)) =
2 ·: 3 ·: sieve (5 ·: [(3 ∤) ◁ [(2 ∤) ◁ ⟨6, [7, [8, [9, ...⟩) =
2 ·: 3 ·: 5 ·: sieve ((5 ∤) ◁ (3 ∤) ◁ [(2 ∤) ◁ ⟨6, [7, [8, [9, ...⟩) =
...
```

Man sieht, dass sich die einzelnen Filter-Operationen ebenfalls in einer lazy Manier aufsammeln und jeweils en bloc ausgeführt werden, wenn das nächste Element gewünscht wird.

De facto baut sich also vor *allNats* die Liste der bisher gefundenen Primzahlen auf, durch die alle folgenden Zahlen „ausgesiebt“ werden. (In der Literatur wird der Algorithmus oft so geschrieben, dass diese Liste vom Programmierer explizit aufgebaut und verwaltet wird.)

2.4.2 Approximationsaufgaben

Eine Klasse von Programmieraufgaben, bei denen lazy Listen besonders handlich sind, betrifft *Approximationsaufgaben*. Bei dieser Art von Problemen hat man im Allgemeinen einen Startwert, von dem aus man eine Folge von Werten generiert, die das gewünschte Resultat immer besser approximieren. Man bricht den Näherungsprozess ab, wenn die Werte sich „stabilisiert“ haben.

Beispiel 2.4 (Quadratwurzel)

Die Berechnung der Quadratwurzel $x = \sqrt{a}$ wird in der Numerischen Mathematik nach dem Verfahren von Newton-Raphson folgendermaßen berechnet. Man verwendet die Tatsache, dass die Zahlenfolge

$$x_0, x_1, x_2, x_3, \ldots \quad \text{mit} \quad x_{i+1} \stackrel{\text{def}}{=} x_i - \frac{f(x_i)}{f'(x_i)}$$

gegen eine Nullstelle der Funktion f konvergiert. Um das für die Berechnung von $x = \sqrt{a}$ auszunutzen, müssen wir also eine Nullstelle folgender Funktion berechnen:

$$f(x) \stackrel{\text{def}}{=} x^2 - a, \quad \text{mit} \quad f'(x) = 2x \quad \text{und} \quad x_{i+1} = \frac{1}{2} \cdot (x_i + \frac{a}{x_i}) \stackrel{\text{def}}{=} h_a(x_i)$$

Wir brauchen also wieder eine unendliche Folge:

$\langle x_0,\ h_a(x_0),\ h_a^2(x_0),\ h_a^3(x_0),\ \ldots \rangle$

Wir hatten weiter vorne (in Abschnitt 2.1) gesehen, dass das mit folgender Definition zu erreichen ist:

```
FUN approx: Real → List Real
DEF approx a = x0 ·: (h a) * (approx a)
               WHERE
               x0 = «geeigneter Startwert»
```

Damit bleibt noch die Frage des *Abbruchkriteriums*. Aufgrund der mathematischen Analyse wissen wir, dass die Folge konvergiert. Das heißt, wir können die Berechnung abbrechen, sobald eine hinreichende Genauigkeit erreicht ist. Wir benutzen dazu die Variante der Funktion Filter, die mit zweistelligen Prädikaten arbeitet.

```
FUN sqrt: Real → Real
DEF sqrt(a) = IF a ≥ 0 THEN ft(≃◁(approx a)) FI
```

Dabei verwenden wir z. B. den Test

```
FUN _ ≃ _: Real × Real → Bool
DEF a ≃ b = |a − b| < 10^−8
```

Übrigens: Den «geeigneten Startwert» erhält man am besten, wenn man den Exponenten der Zahl a halbiert. Denn nach dem IEEE-Standard ist eine

Gleitpunktzahl im Wesentlichen dargestellt in der Form $1.xxxx \cdot 2^e$, und es gilt $(1.xxxx \cdot 2^e) \approx (1.0 \cdot 2^{\frac{e}{2}})^2$. Die Extraktion und Halbierung des Exponenten muss durch geeignete Shift-Operationen auf Bit-Ebene bewerkstelligt werden. Leider stellt so gut wie keine Programmiersprache eine solche Operation „Exponent von ...“ bereit.

Beispiel 2.5 (Differenziation)

Ähnlich zur Berechnung der Quadratwurzel verlaufen auch andere Approximationsverfahren. Ein weiteres einfaches Beispiel ist die Berechnung des Wertes der *Ableitung* einer beliebigen („gutartigen“) Funktion f an einer Stelle x_0. Wir haben – für hinreichend kleines h – die Näherungsformel

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Die Idee ist dann ganz einfach, eine Folge von immer kleineren h-Werten zu nehmen, bis der obige Wert sich stabilisiert. Eine geeignete Folge ist

$$\langle h, \frac{h}{2}, \frac{h}{4}, \frac{h}{8}, \frac{h}{16}, \ldots \rangle$$

Das lässt sich mit unseren Mitteln ganz einfach programmieren, wobei wir die Hilfsfunktionen aus dem Wurzelbeispiel entsprechend wiederverwenden.

```
FUN diff: (Real → Real) → Real → Real
DEF (diff f) x = ft (≃ ◁ (D f x) * hList)
```

Dabei verwenden wir die Hilfsfunktion

```
FUN D: (Real → Real) → Real → Real → Real
DEF D f x h = (f(x+h)−f(x−h)) / (2·h)
```

und die Liste

```
FUN hList: List Real
DEF hList = h0 ·: (_ / 2) * hList
            WHERE
            h0 = «geeigneter Startwert»
```

2.4.3 Animation („Ströme“)

Es ist bekannt, dass in der Funktionalen Programmierung die Eleganz etwas leidet, wenn man mit Ein-/Ausgabe arbeiten muss. Das ist besonders dann kritisch, wenn der funktionale Algorithmus mit der Ein-/Ausgabe *verschränkt* werden soll. Ausführlich werden wir dieses Thema erst im Kapitel 17 behandeln. Aber für eine gewisse Klasse von Aufgaben kann eine einfache Lösung mit Hilfe von lazy Listen erreicht werden. In dieser Anwendung – also bei

Ein-/Ausgabe – wird insbesondere auch oft der andere Name für lazy Listen verwendet: *Ströme* (engl.: *streams*).

Beispiel 2.6 (8-Queens mit Animation)

Das *8-Queens-Problem* ist wohlbekannt: Man setze 8 Damen so auf ein Schachbrett, dass keine eine andere bedroht (s. Abbildung 20.1 auf Seite 421). Das ist eine standardmäßige Backtrack-Aufgabe, die sich funktional ganz kurz folgendermaßen schreiben lässt. (Wir arbeiten hier mit der Variante der *potenziell unendlichen Listen*.)

```
FUN queens: Configuration → List Configuration
DEF queens(pConf) =                          -- pConf ≙ "partial configuration"
    IF |pConf| = 8 THEN ⟨pConf⟩
    IF |pConf| < 8 THEN ++ /(queens ∗ ( legal ◁ (pConf:·) ∗ (1..8) ))
    FI
```

Wir wollen die Liste aller Lösungen ausgeben. Dazu bauen wir partielle Konfigurationen auf, bei denen eine Anzahl von $i \leq 8$ Damen gesetzt sind. Falls $i < 8$ hängen wir an die Konfiguration alle möglichen Positionen für die $(i+1)$te Dame an. Die so entstehende Liste neuer Konfigurationen wird dann durch das Prädikat *legal* gefiltert. Dieses Prädikat prüft eine Konfiguration auf gegenseitige Bedrohungen der Damen durch gleiche Zeilen und Diagonalen. Auf jede verbleibende legale (partielle) Konfiguration wird dann rekursiv *queens* wieder angewendet. Dadurch entsteht eine Liste von Listen, die durch Konkatenation wieder flach gemacht wird. (In Kapitel 16 werden wir diesen Algorithmus intensiver studieren.)

Dabei verwenden wir einige Hilfsfunktionen, deren Programmierung wir hier nicht explizit hinschreiben, etwa $(1..8) = \langle 1,2,3,4,5,6,7,8\rangle$, das Prädikat *legal* oder die Operation „ :· “ zur Erweiterung von Konfigurationen.

Interessanter ist für uns jetzt die Animation. Das heißt, jede Lösung soll grafisch ausgegeben werden. (Die tatsächliche grafische Aufbereitung interessiert uns hier nicht; das wird erst in Kapitel 20 diskutiert. Im Augenblick geht es uns nur um die Interaktion zwischen Algorithmus und I/O-Komponente.) Wir gehen also von einer Ausgabefunktion folgender Bauart aus, wobei der Typ *Action* für Ausgabeaktionen steht und der Operator & für deren sequenzielle Ausführung, was funktional eine Form der Funktionskomposition ist (s. Kapitel 17 und 18):

```
FUN animate: List Configuration → Action
DEF animate ◊ = done
DEF animate(config ·: rest) = show(config) & animate(rest)
```

Warum ist in diesem Beispiel – das nur ein paar Dutzend Lösungen hat – die Verwendung einer lazy Liste besser als die Verwendung von Sequenzen?

- Bei Sequenzen würden zunächst *alle* Lösungen berechnet, bevor mit der Ausgabe der ersten begonnen würde. Das macht sich für den Benutzer in einer deutlichen Verzögerung der Startzeit bemerkbar.
  Im obigen Design wird dagegen jeweils nur die nächste gewünschte Lösung berechnet und sofort gezeigt.
- Mit entsprechenden Erweiterungen des obigen Programms kann man leicht interaktive Modifikationen einbauen. Dann kann sich der Benutzer z. B. auch partielle Konfigurationen zeigen lassen.

Was allerdings bei beiden Varianten völlig fehlt, ist eine parallele Verarbeitung: Während der Benutzer eine Lösung ansieht, rechnet das System die nächste aus. Dazu brauchen wir andere Programmiermittel (s. Kapitel 19).

3

Parser als Funktionen höherer Ordnung

Die Grammatik ist die Experimentalphysik der Sprachen.
A. de Rivarol (Maximen und Gedanken)

Es gehört zur Folklore der *Functional-programming community*, dass sich durch Verwendung von Funktionen höherer Ordnung Scanner und Parser besonders klar und konzise schreiben lassen. Diese Folklore ist auch in einer ganzen Reihe von Publikationen aufbereitet (unter anderem [85, 86, 76, 108, 88, 136, 112]).

Man mag nun einwenden, dass das alles fruchtlose Fingerübungen sind, denn schließlich gibt es Parser und sogar Parsergeneratoren in reichlicher Anzahl. Und die sind auch meistens effizienter als die Parser, die wir als Funktionale schreiben. Trotzdem lohnt es sich:

Zum einen sind Fingerübungen etwas sehr Nützliches. Zum anderen ist die Möglichkeit, selbst schnell und einfach Parser schreiben zu können, auch ungeheuer praktisch:

- Solche Parser können als Prototypen dienen, mit denen man Experimente zur geplanten Sprache durchführt, bevor man sich an die Arbeit zum endgültigen Werkzeug macht.
- Man braucht nicht den Formalismus des jeweiligen Generators zu lernen, sondern bleibt innerhalb der eigenen Programmiersprache.
- Da man in der Programmiersprache bleibt, kann man besonders leicht und flexibel die so genannten semantischen Aktionen programmieren, die zu jedem Parser gehören. (Hier bieten viele Generatoren nur äußerst eingeschränkte Möglichkeiten.)
- Viele Parser und Generatoren verlangen starke Restriktionen beim Sprachdesign, um Effizienz zu garantieren. Wenn eine Sprache diesen Restriktionen nicht genügt, muss man sich mühsam um die Klippen „herummogeln“.
- Ein neuerer Aspekt ist, dass man „eingebettete“ Sprachen hat, das heißt, innerhalb eines Programms der Sprache X gibt es Fragmente, die zur Spra-

che Y gehören. Hier muss man flexibel zwischen unterschiedlichen Parsern umschalten können.

- Außerdem sind die heutigen Rechner so schnell geworden, dass die Effizienzvorteile der „schnellen“ Parser praktisch gar nicht mehr feststellbar sind. Der „Prototyp-Parser“ kann dann gleich als Produkt dienen.

Insgesamt erhalten wir die Architektur von Abbildung 3.1. Es gibt zwei primäre Strukturen *MetaParser* und *MetaScanner*, die als generelle Werkzeuge in der Bibliothek hinterlegt werden. Auf diesen aufbauend werden dann für spezielle Grammatiken die entsprechenden Parser und Scanner definiert.

Die Strukturen *MetaParser* und *MetaScanner* sind generisch programmiert, so dass sie für spezielle Grammatiken mit den entsprechenden Typen für die so genannten Token und Syntaxbäume instanziiert werden können. Wir werden die Beispiele im Folgenden etwas vereinfachen, indem wir anstelle der Token (die noch Attribute wie Position etc. enthalten) nur einfache Strings verwenden.

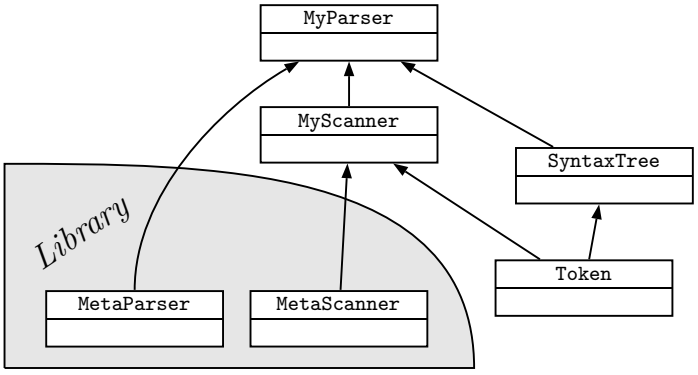


Abb. 3.1: Verwendung der Kombinator-Bibliothek

In den folgenden Abschnitten beschäftigen wir uns mit der Programmierung der beiden Strukturen *MetaParser* und *MetaScanner*.

Es gibt unzählige Varianten von Parsern, mit und ohne Fehlerbehandlung, mit vollem und mit partiellem Backtrack, mit und ohne Optimierung, in monadischer Form und im so genannten Arrow-Stil, für beliebige kontextfreie Sprachen und für eingeschränkte Sprachklassen usw. Da wir hier nicht an Compilerbau interessiert sind, sondern an Prinzipien der Funktionalen Programmierung, beschränken wir uns auf eine programmiertechnisch besonders einfache Form von Parsern, die nur für gewisse eingeschränkte Sprachklassen anwendbar sind (im Wesentlichen auf so genannte LL(1)-Grammatiken). Auf mögliche Verallgemeinerungen gehen wir am Ende des Kapitels in Abschnitt 3.4 noch kurz ein.

3.1 Vorbemerkung zu Grammatiken und Syntaxbäumen

Wir setzen hier elementares Grundwissen über Formale Sprachen und Grammatiken voraus, insbesondere die Begriffe Produktionsregel, Terminal- und Nichtterminalzeichen (siehe etwa [80]). Wir werden im Folgenden grundsätzlich mit *kontextfreien Grammatiken* arbeiten, also Grammatiken, deren Produktionsregeln von folgender Form sind:

$$A \to u$$

mit einem Nichtterminal A und einem String $u \in V^*$, wobei das Alphabet $V = T \cup N$ sich aus den Terminalzeichen T und den Nichtterminalzeichen N zusammensetzt.

Zur Illustration der folgenden Diskussionen verwenden wir ein möglichst kurzes, aber illustratives Beispiel, das in Tabelle 3.1 angegeben ist. Diese Grammatik erzeugt Zeichenreihen der Art `"* * * xy"`, also Identifier, denen ein oder mehrere Sterne vorausgehen.

`A`	$\to$	`S Ide` $[\widetilde{a_1}]$
`S`	$\to$	`"*" S` $[\widetilde{s_1}]$
`S`	$\to$	$[\widetilde{s_2}]$
`Ide`	$\to$	`...`

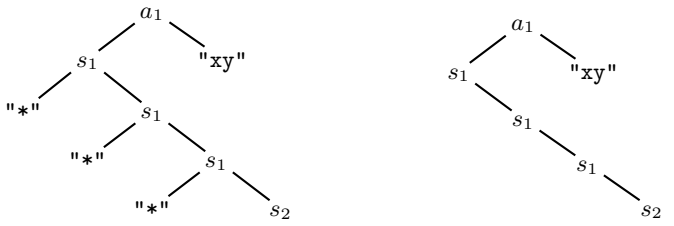
Tab. 3.1: Eine Beispielgrammatik

Den Aufbau von Identifiern und die Behandlung von Leerzeichen zwischen den Elementen ignorieren wir für den Augenblick, um die Diskussion knapp und konzise zu halten. Die Bedeutung der Symbole $\widetilde{a_1}$, $\widetilde{s_1}$ und $\widetilde{s_2}$ werden wir weiter unten erläutern. Man beachte, dass die dritte Produktion eine leere rechte Seite hat (weil das Symbol $\widetilde{s_2}$ keine Zeichen erzeugt).

Das *Parsieren* eines Strings der Bauart `"* * * xy"` nach dieser Grammatik führt dann auf einen so genannten *Syntaxbaum*. Dabei gibt es zwei Varianten, die in Abbildung 3.2 illustriert sind: *konkrete* und *abstrakte Syntaxbäume*. Bei den ersteren bleiben auch konstante Terminalzeichen erhalten, während diese bei den letzteren zur Steigerung der Speichereffizienz weggelassen werden. Im Folgenden arbeiten wir mit konkreten Syntaxbäumen.

Damit wird auch die Rolle der Symbole a_1, s_1 und s_2 in der Grammatik aus Tabelle 3.1 klar: Sie sind die Konstruktoren der entsprechenden Syntaxbäume. Die Varianten $\widetilde{a_1}$ etc. binden diese Konstruktorfunktionen als so genannte *semantische Aktionen* in die Grammatik ein.

Zunächst definieren wir den Typ der Syntaxbäume. Auch hier müssen wir eine Designentscheidung treffen: Am einfachsten ist es, einen einheitlichen Typ *Tree* für alles zu verwenden; man kann aber auch für jedes Nichtterminal einen eigenen Untertyp von *Tree* einführen. Die zweite Variante macht zwar etwas



konkreter Syntaxbaum abstrakter Syntaxbaum

Abb. 3.2: Syntaxbäume

mehr Aufwand in der Programmierung, ist dafür aber typsicherer, weshalb wir sie als überlegen ansehen.

```
STRUCTURE SyntaxTree = {
  TYPE ATree  =  a1(STree × Ide)      -- konkrete Syntaxbäume zu A
  TYPE STree  =  s1(String × STree)   -- konkrete Syntaxbäume zu S
              |  s2
  TYPE Tree   =  ATree | STree        -- alle konkreten Syntaxbäume
  TYPE Ide    =  ide(symbol: String)  -- Identifier
}
```

Wie man beim Vergleich mit der Grammatik in Tabelle 3.1 sieht, ergibt sich die Struktur der Baumtypen unmittelbar aus der zugehörigen Grammatik.

3.2 Parser

Ein *Parser* ist im Wesentlichen eine Funktion, die als Argument einen String nimmt und als Ergebnis einen zugehörigen (abstrakten oder konkreten) Syntaxbaum abliefert. Da aber nicht auszuschließen ist, dass der String fehlerhaft ist, muss auch die Möglichkeit des Scheiterns eingeplant werden. Deshalb ist das Ergebnis nicht *Tree*, sondern *Maybe Tree*, wobei wir den folgenden Hilfstyp verwenden (s. Abschnitt 8.1.1):

```
TYPE Maybe α = α | { fail }
```

Eine weitere Komplikation entsteht dadurch, dass nur beim Aufruf der äußersten Parsingfunktion der gesamte String „verbraucht“ wird; im Allgemeinen verarbeiten die Parsingfunktionen nur einen gewissen Initialteil des gegebenen Strings. Das sieht man sehr leicht an unserem Beispiel `"* * * xy"`. Die Funktion, die die drei Sterne parsiert, lässt als Rest den String `"xy"` übrig. Diese Überlegungen führen dazu, folgenden polymorphen Typ für Parser einzuführen:

```
TYPE Parser α = (String → Maybe(α) × String)
```

Dabei steht die Typvariable α für die verschiedenen Typen von Syntaxbäumen, in unserem Beispiel also für *ATree* und *STree*, die als erstes Ergebnis abgeliefert werden. Das zweite Ergebnis ist der verbliebene Reststring.

Unser Ziel ist es jetzt, Operatoren einzuführen, mit deren Hilfe wir zu einer gegebenen Grammatik ganz leicht den zugehörigen Parser programmieren können. Zur Motiven zeigen wir zunächst, wie – unter Verwendung der gleich zu definierenden Operatoren – unsere Beispielgrammatik aus Tabelle 3.1 in einen Parser umgesetzt werden kann.

```
STRUCTURE MyParser = {
  IMPORT  MetaParser      -- die generellen Parserfunktionale
          SyntaxTree      -- die Syntaxbäume (ATree, STree)
          MyScanner       -- Analyse von Identifiern (Ide)
  FUN A: Parser(ATree)
  FUN S: Parser(STree)
  FUN Ide: Parser(String)
  DEF A = S ; Ide ; ã1
  DEF S = "*" ; S ; s̃1
        | s̃2
}
```

Wie man sieht, ist der Parser eine direkte Eins-zu-Eins-Umsetzung der Grammatik. (Da wir unsichtbare Operatoren zulassen, könnten wir sogar den Operator „ ; “ weglassen, sodass die Grammatik praktisch identisch zum Parser wäre.) Diese Anwendung illustriert die Verwendung der Parsing-Operatoren, die in Programm 3.1 definiert sind.

Aufgrund der Typisierung der einzelnen Operatoren ergibt sich im obigen Beispiel z. B. die Klammerung

```
DEF S = (("*" ; S) ; s̃1) | s̃2
```

wobei für s_2 noch ein entsprechendes Lifting nach *Parser*(*STree*) ergänzt wird.

Die Typen und Funktionen zum Parsieren werden in einer Struktur *MetaParser* zusammengefasst. Neben dem primären Typ *Parser* verwenden wir noch einen Hilfstyp *Action* zur Darstellung der semantischen Aktionen. Dieser Typ dient im Wesentlichen dazu, die verschiedenen Konstruktorfunktionen der Baumtypen einheitlich einzupacken. (Dieser Typ wäre nicht unbedingt erforderlich; er verbessert aber die Lesbarkeit.)

Der erste Operator $(A\,;B)$ verbindet zwei Parserfunktionen sequenziell. Bei der Anwendung wird zuerst A auf den gegebenen String s angewandt. Im einfachsten Fall entsteht daraus ein Baum a und es bleibt ein Reststring r. Auf diesen wird dann der Parser B angewandt, der im Erfolgsfall einen Baum b und einen weiter verkürzten Reststring r' liefert. Das Gesamtergebnis besteht dann aus dem Paar $(a,\ b)$ und dem Reststring r'. Im Allgemeinen können aber durch die Teilparser A und B nicht nur einzelne Bäume, sondern ganze Tupel von Bäumen erzeugt werden. Deshalb brauchen wir die assoziati-

Programm 3.1 Die Parsing-Operatoren

```
STRUCTURE MetaParser = {
  TYPE Parser α = (String → Maybe(α) × String)
  TYPE Action α = action(cons: α)

  FUN _ ; _: Parser(α) × Parser(β) → Parser(α ⊗ β)
  FUN _ ; _: Parser(α) × Action(α → β) → Parser(β)
  FUN _ | _: Parser(α) × Parser(α) → Parser(α)
  FUN ~_: α → Action(α)
  FUN ~_: α → Parser(α)                                   -- Lifting
  FUN _: String → Parser(String)                          -- Scanner

  DEF (A ; B)(s) = LET (a, r) = A(s) IN
                   IF a ≠ fail THEN LET (b, r') = B(r) IN
                                    IF b ≠ fail THEN (a ⊗ b, r')   -- Erfolg
                                    IF b = fail THEN (fail, s) FI  -- Backtrack
                   IF a = fail THEN (fail, s)             -- Backtrack
                   FI

  DEF (A ; action(f))(s) = LET (a, r) = A(s) IN
                           IF a ≠ fail THEN (f(a), r)     -- Erfolg
                           IF a = fail THEN (fail, s) FI  -- Backtrack

  DEF (A | B)(s) = LET (a, r) = A(s) IN
                   IF a ≠ fail THEN (a, r)                -- Erfolg
                   IF a = fail THEN B(s) FI               -- Backtrack

  DEF ~f: Action = action(f)
  DEF ~a: Parser = λ s • (a, s)                           -- Lifting

  DEF (terminal: String): Parser = shift(terminal)        -- Lifting

  FUN shift: String → Parser(String)
  DEF shift(t)(s) = ...                                   -- Scanner
}
```

ve Komposition von Tupeln, für die z. B. $(a, b) \otimes (c, d, e) = (a, b, c, d, e)$ gilt (s. Abschnitt 6.4.1).

Falls einer der beiden Parser A oder B scheitert, wird insgesamt das Ergebnis *fail* abgeliefert und der Eingabestring s unverändert durchgereicht. Damit wird (zumindest partielles) Backtracking erreicht.

Am Ende jeder Produktion steht eine semantische Aktion $\widetilde{f}$, die aus einem Konstruktor f des entsprechenden Baumtyps abgeleitet ist. Diese Aktion wird mit einem Parser in der Form $(A\,;\widetilde{f})$ zu einem neuen Parser verbunden. Bei der Anwendung auf einen String s wird zunächst A auf s angewandt. Das liefert im Allgemeinen ein Tupel $(a_1, \ldots, a_n)$ von Bäumen, auf die dann der Konstruktor f angewandt wird, um einen neuen Baum $a = f(a_1, \ldots, a_n)$ zu erzeugen. Dieser Baum und der Reststring r werden dann abgeliefert. Auch hier wird beim Scheitern des Parsers A wieder *fail* abgeliefert und Backtrack ermöglicht.

Der Operator $(A \mid B)$ erlaubt die Auswahl zwischen zwei Parsern. Idealerweise sollte diese Operation symmetrisch sein, was aber technisch nur schwer machbar wäre. Deshalb arbeitet unsere einfache Implementierung sequenziell: Zuerst wird der Parser A versucht. Falls er erfolgreich ist, bestimmt er auch das Gesamtergebnis. Ansonsten wird der Parser B genommen.

Der Operator $\widetilde{f}$ dient nur dazu, einen Baumkonstruktor zu einer semantischen Aktion zu machen, um so die Typisierung der anderen Operationen zu erleichtern. Bei Bedarf – z. B. bei der leeren Produktion $\widetilde{s_2}$ im obigen Beispiel – muss gleich ein Lifting zu einem Parser stattfinden, der den Baum a und den unveränderten String s abliefert.

Um das Schreiben der Grammatiken besonders bequem zu machen, erlauben wir, Terminalzeichen direkt als Strings anzugeben. Deshalb brauchen wir einen (unsichtbaren) Casting-Operator, der aus einem String einen Parser macht. Die Funktion *shift* prüft, ob das Terminaltoken t in der Tat den Anfang des Eingabestrings s bildet, also $s = t +\!\!+ r$ gilt; falls ja, werden dieser String t und der verbleibende Reststring r abgeliefert, ansonsten *fail* und s selbst. (Mehr zu Scannern diskutieren wir gleich in Abschnitt 3.3.)

Diese Art der Parserprogrammierung ist sehr einfach, hat aber auch starke Einschränkungen bzgl. der Anwendbarkeit. Darauf gehen wir in Abschnitt 3.4 noch einmal kurz ein.

3.3 Scanner

In den Beispielen des vorigen Abschnitts haben wir das Erkennen von Identifiern offen gelassen. Diese Art von Aufgaben wird traditionell nicht von Parsern, sondern von *Scannern* erledigt. Diese basieren üblicherweise nicht auf den kontextfreien Grammatiken, sondern auf den einfacheren regulären Grammatiken. Das ist heute aber aus mehreren Gründen veraltet:

- Angesichts der Geschwindigkeit und der Speicherkapazität heutiger Rechner ist der minimale Effizienzgewinn den zusätzlichen Programmieraufwand nicht wert.
- Viele Aspekte – z. B. geschachtelte Kommentare – übersteigen die Mächtigkeit regulärer Grammatiken, so dass sie von traditionellen Scannern ohnehin nicht verarbeitet werden können.
- Moderne Programmiersysteme vermischen Dokumentationssprachen wie LaTeX mit Programmiersprachen, so dass man flexibel zwischen verschiedenen Scannern umschalten muss.

Aus diesen Gründen führen wir für Scanner keine gesonderten Parsingkonzepte ein, sondern versuchen unsere bisherigen Mechanismen weitgehend beizubehalten. Wir illustrieren das Vorgehen anhand der Grammatik in Tabelle 3.2, die die fehlenden Identifier zur Grammatik in Tabelle 3.1 beschreibt.

Zu dieser Grammatik assoziieren wir das folgende Programm (das auf den Operatoren von Programm 3.2 basiert):

```
Ide     →  Letter Ide [ide1~]
Ide     →  Letter [ide2~]
Letter  →  (("a".."z") | ("A".."Z")) [letter~]
```

Tab. 3.2: Beispielgrammatik (Fortsetzung)

```
STRUCTURE MyScanner = {
  IMPORT MetaScanner
  FUN Ide: Scanner
  FUN Letter: Scanner
  DEF Ide = (Letter ; Ide) | Letter
  DEF Letter = ("a".."z") | ("A".."Z")
}
```

Man beachte, dass die Reihenfolge bei der Definition von *Ide* essenziell ist; denn die rekursive Regel muss vor der Abbruchregel ausgeführt werden. Dazu betrachte man einen Identifier `"abc"`, dem z. B. ein Leerzeichen folgt. Nach mehreren Inkarnationen von *Ide* kommt das Leerzeichen; damit wird in dieser Inkarnation der linke Zweig zu *fail*, sodass dann der rechte Zweig genommen wird, was letztlich zur erfolgreichen Erkennung von `"abc"` führt. Damit wird das Prinzip des so genannten *longest match* realisiert. Allerdings funktioniert das nicht generell so einfach wie in diesem Beispiel.

Auch der Scanner basiert auf vordefinierten Funktionen höherer Ordnung, die in Programm 3.2 angegeben sind. Diese Funktionen folgen im Prinzip den gleichen Konzeptionen wie die entsprechenden Parserfunktionen. Da wir jedoch nicht unterschiedliche Arten von Bäumen erzeugen, sondern immer nur Strings miteinander konkatenieren, können wir auf die Angabe unterschiedlicher semantischer Aktionen verzichten.

Unser Programm 3.2 zeigt nur das wesentliche Grundprinzip. Für einen praktikablen Scanner wären noch einige technische Details hinzuzufügen, auf die wir hier jedoch verzichten.

Das gilt insbesondere für die so genannten Token, auf die wir hier verzichten; deshalb ist die Architektur unseres Programms gegenüber Abbildung 3.1 etwas vereinfacht. Üblicherweise werden die von Scannern erkannten Strings nicht direkt zurückgeliefert, sondern zuvor noch mit weiteren Informationen angereichert, z. B. mit der Position im Programmtext (für Fehlermeldungen). Außerdem werden die Strings oft noch mit Hilfe einer so genannten Symboltabelle durch Indizes ersetzt, wodurch die weiteren Phasen des Compilers etwas effizienter werden.

Programm 3.2 Die Scanner-Operatoren

```
STRUCTURE MetaScanner = {
  TYPE Scanner = (String → Maybe(String) × String)
  FUN _ ; _: Scanner × Scanner → Scanner
  FUN _ | _: Scanner × Scanner → Scanner
  FUN _: Char → Scanner
  FUN _..._: Char × Char → Scanner
  DEF (A ; B)(s) = LET (a, r) = A(s) IN
                   IF a ≠ fail THEN LET (b, r') = B(r) IN
                                    IF b ≠ fail THEN (a ++ b, r')   -- Erfolg
                                    IF b = fail THEN (fail, s) FI  -- Backtrack
                   IF a = fail THEN (fail, s)                      -- Backtrack
                   FI
  DEF (A | B)(s) = LET (a, r) = A(s) IN
                   IF a ≠ fail THEN (a, r)                         -- Erfolg
                   IF a = fail THEN B(s) FI                        -- Backtrack
  DEF ((c: Char): Scanner)(s) = IF ft(s) = c THEN (ft(s), rt(s))   -- Erfolg
                                IF ft(s) ≠ c THEN (fail, s) FI     -- Backtrack
  DEF (c1..c2)(s) = IF c1 ≤ ft(s) ≤ c2 THEN (ft(s), rt(s))        -- Erfolg
                                        ELSE (fail, s) FI         -- Backtrack
}
```

3.4 Verallgemeinerungen

Die Parsingtechnik, die in Programm 3.1 in Abschnitt 3.2 beschrieben ist, unterliegt starken Einschränkungen, die man auf unterschiedliche Art und Weise beheben kann. Wir betrachten einige Beispiele.

Volles Backtracking

Als Erstes betrachten wir die folgende Grammatik:

`A`	→	`(B | C) "z"`	$[\widetilde{a_1}]$
`B`	→	`"x"`	$[\widetilde{b_1}]$
`C`	→	`"x" "y"`	$[\widetilde{c_1}]$

Tab. 3.3: Eine problematische Beispielgrammatik

Wenn wir den zugehörigen Parser auf die Eingabe `"x y z"` anwenden, dann erhalten wir *fail*, weil zuerst der Parser B angewandt wird, der mit einem Erfolg endet. Der verbliebene Parser `"z"` wird dann auf den restlichen Eingabestring `"y z"` angewandt, was scheitert.

Diese Art von Problemen wird behoben, wenn man statt unserer partiellen Backtrack-Technik ein *volles Backtracking* vorsieht. Dazu muss der Typ *Parser* entsprechend komplexer werden:

$$\text{TYPE } \mathit{Parser}(\alpha) = (\mathit{String} \to \mathit{Set}(\alpha \times \mathit{String}))$$

Diese Art von Parsern berechnet jetzt nicht mehr einen Baum bzw. *fail*, sondern die Menge aller möglichen Bäume und ihrer zugehörigen Reststrings. Damit wird die Auswahl sehr einfach programmierbar als Mengenvereinigung:

$$\text{DEF } (A \mid B)(s) = A(s) \cup B(s)$$

Die sequenzielle Komposition von Parsern wird jetzt aber wesentlich komplexer:

$$\begin{aligned}\text{DEF } (A\,;B)(s) = {} & \cup/(\varphi(B) * A(s)) \\ & \text{WHERE} \\ & \varphi(B)(a,r) = \psi(a) * B(r) \\ & \qquad\qquad \text{WHERE} \\ & \qquad\qquad \psi(a)(b,r') = (a \otimes b, r')\end{aligned}$$

Zuerst wird der Parser A angewandt, was eine (möglicherweise leere) Menge von Ergebnissen – also Bäume oder Baumtupel zusammen mit ihren Reststrings – liefert. Für jedes dieser Paare (a, r) wird jetzt mittels der Hilfsfunktion φ zunächst der Parser B auf r angewandt und dann das Ergebnis mit a konkateniert. Da aber auch B eine ganze Menge von Ergebnissen produzieren kann, müssen wir dies elementweise tun. Insgesamt erhalten wir also eine Menge von Mengen, die mittels $\cup/$ wieder zu einer Menge verschmolzen werden muss.

Mit einem solchen vollen Backtrack-Parser kann man zwar prinzipiell alle kontextfreien Sprachen verarbeiten, aber es bleiben einige Probleme:

- Der Parser ist im Allgemeinen relativ ineffizient.
- Wenn der Eingabestring Fehler enthält, dann liefert der Parser nur die leere Menge als Zeichen des Scheiterns. Er gibt keinen Hinweis, *wo* der Fehler lag. (Dazu muss man zusätzlichen Programmieraufwand treiben.)
- Grammatiken mit *linksrekursiven Produktionen* sind nicht unmittelbar verarbeitbar. Diesen Punkt betrachten wir gleich noch genauer.

Linksrekursion

Zur Illustration betrachten wir die Grammatik in Tabelle 3.4, die Funktionsapplikationen in Curry-Form beschreibt, also Terme der Art `"f x y z"`, wobei folgende implizite Klammerung erreicht werden muss: `"(((f x) y) z)"`.

Wenn man zu dieser Grammatik ganz naiv den zugehörigen Parser konstruieren würde, dann würde dieser nicht terminieren:

$$\text{DEF } A = (A \;;\; \mathit{Ide}) \mid \mathit{Ide} \quad -\!- \textit{ nichtterminierender Parser}$$

`A`	$\rightarrow$	`A Ide` $[\widetilde{a_1}]$
`A`	$\rightarrow$	`Ide` $[\widetilde{a_2}]$

Tab. 3.4: Eine linksrekursive Beispielgrammatik

Die Rettung aus diesem Dilemma besteht in einer geeigneten Transformation der Grammatik. So ist z. B. die Grammatik in Tabelle 3.4 gleichwertig zu der in Tabelle 3.5 (mit einem neuen Nichtterminal `A'`). Man beachte, dass dabei eine Produktion mit leerer rechter Seite entsteht (was traditionell mit dem Buchstaben ε geschrieben wird).

`A`	$\rightarrow$	`Ide` $[\widetilde{a_2}]$ `A'`
`A'`	$\rightarrow$	`Ide` $[\widetilde{a_1}]$ `A'`
`A'`	$\rightarrow$	

Tab. 3.5: Die transformierte Grammatik aus Tabelle 3.4

In dieser transformierten Grammatik treten allerdings einige Komplikationen auf. Weil die semantischen Aktionen nicht mehr notwendigerweise am Ende der Produktionen stehen, müssen wir unsere Operatoren anpassen. Außerdem muss z. B. die Funktion A' jetzt als Argument den bereits parsierten Baum bekommen. Zur Grammatik in Tabelle 3.5 gehören damit die folgenden Funktionalitäten und Definitionen:

FUN A: $Parser(ATree)$
FUN A': $ATree \rightarrow Parser(ATree)$
DEF $A = Ide \ ; \ \widetilde{a_2} \ ; \ A'$
DEF $A'(a) = (a \ ; \ Ide \ ; \ \widetilde{a_1} \ ; \ A') \mid a$

Damit dies funktioniert, müssen wir noch eine weitere Variante der sequenziellen Komposition einführen:

FUN $\_ \ ; \ \_$: $Parser(\alpha) \times (\alpha \rightarrow Parser(\beta)) \rightarrow Parser(\beta)$

Die Definition dieser Variante überlassen wir dem interessierten Leser als Übung.

Effizienzsteigerung

Mit weiteren Transformationen – vor allem mit der so genannten *Linksfaktorisierung* – lässt sich die Effizienz der Parser steigern, weil Backtracking seltener nötig wird oder Sackgassen früher erkannt werden. Ein Beispiel zeigt die Grammatik in Tabelle 3.6, die durch Linksfaktorisierung aus Tabelle 3.3 hervorgeht.

A	→	"x" (B′ | C′) "z" $[\widetilde{a_1}]$
B′	→	$[\widetilde{b_1}]$
C′	→	"y" $[\widetilde{c_1}]$

Tab. 3.6: Die umgeformte Problemgrammatik aus Tabelle 3.3

Dies lässt sich noch mit der Idee der so genannten First- und Followmengen verbinden (auf die wir hier nicht näher eingehen können). Diese Technik liefert für jede Produktion „Wächter“, das heißt Mengen von Terminalzeichen, die am Anfang des Eingabestrings stehen müssen, wenn die Produktion anwendbar ist. Diese Wächter liefern also eine notwendige (aber keine hinreichende) Bedingung für die Anwendbarkeit der Produktion. (Wenn diese Wächter disjunkt sind, spricht man von LL(1)-Grammatiken.) Das folgende Programm realisiert dieses Prinzip für die Grammatik aus Tabelle 3.6, wobei wir einen entsprechend definierten Operator

FUN $\_ \Rightarrow \_$: *Set Char* × *Parser*(α) → *Parser*(α)
DEF $(G \Rightarrow A)(s)$ = IF $ft(s) \in G$ THEN $A(s)$ ELSE $(fail, s)$ FI

voraussetzen, der zur Struktur *MetaParser* hinzugefügt werden müsste.

FUN A: *Parser*(*ATree*)
FUN B': *String* → *Parser*(*BTree*)
FUN C': *String* → *Parser*(*CTree*)
DEF A = ({"x"} ⇒ "x" ; (B' | C') ; "z" ; $\widetilde{a_1}$)
DEF $B'(x)$ = ({"z"} ⇒ x ; $\widetilde{b_1}$)
DEF $C'(x)$ = ({"y"} ⇒ x ; "y" ; $\widetilde{c_1}$)

Ohne diese Wächter würde z. B. bei der Eingabe "x y z" der Parser B' auf den String "y z" angewandt und seine Aktion $\widetilde{b_1}$ ausführen. Danach würde die Fortsetzung des Parsers A endgültig scheitern. Mit dem Wächter scheitert B' und die (korrekte) Alternative C wird erfolgreich angewandt.

Mit diesen Erweiterungen funktioniert jetzt unser Parser mit partiellem Backtrack auch für diese Grammatik.

Auch wenn man zeigen kann, dass mit solchen Transformationen ein mächtiges und gleichzeitig effizientes Parsingkonzept erreicht werden kann [112], so bleibt doch ein wesentlicher Nachteil: Die Anwendung der Transformationen ist im Allgemeinen so komplex, dass man bereits wieder entsprechende Werkzeuge zu Hilfe nehmen muss. Damit ist der Charme unserer ursprünglichen Idee „*Grammatik = Parser*“ weitgehend verloren. Deshalb sind unsere hier skizzierten Prinzipien vor allem bei den einfacheren Grammatiken angebracht, die keine weiteren Adaptionen brauchen.[1]

[1] Erfreulicherweise sind viele wichtige Applikationsfelder so einfach, dass diese Art von Grammatiken und Parsern ausreicht. Prominente Beispiele dafür sind XML und HTML.

4

Gruppen: Die Basis der Modularisierung

Heil'ge Ordnung, segensreiche
Himmelstochter, die das Gleiche
Frei und leicht und freudig bindet
Schiller (Lied von der Glocke)

Eigentlich gibt es nur zwei fundamentale Konstruktionsprinzipien für (funktionale) Programme. Das erste ist wohlbekannt und wurde in den letzten Kapiteln schon behandelt: Funktionen. Das zweite ist etwas schwieriger zu charakterisieren, denn es tritt in unterschiedlichen Spielarten auf. Deshalb gibt es auch keinen einheitlichen Begriff dafür. Letztlich geht es aber immer darum, eine Reihe von Dingen – z. B. Werte, Funktionen, Typen – zu einem größeren Ganzen zusammenzufassen. Deshalb bieten sich Begriffe wie *Ansammlung*, *Kollektion*, *Gruppe* oder Ähnliches an. Weil es das kürzeste dieser Wörter ist, wählen wir ***Gruppe***.

Anmerkung 1: Viele Programmiersprachen leiden an einer Krankheit: Featuritis. *Sie umfassen ein Konglomerat von Konzepten und Notationen, die ihre Designer aus dem einen oder anderen Grund für nützlich hielten. Nun ist nichts dagegen einzuwenden, dass eine Sprache Ausdrucksmittel bereitstellt, die das Programmieren erleichtern. Im Gegenteil, das Programmieren zu erleichtern ist eine der primären Daseinsberechtigungen von Sprachen. Aber es darf kein zufälliges Konglomerat von individuellen* Features *entstehen; vielmehr muss ein homogenes Design erkennbar sein. Eine wesentliche Vorbedingung dafür ist, dass die einzelnen Features auf einem gemeinsamen, wohl definierten Fundament basieren.*

Anmerkung 2: Auch das elementarste Fundament braucht eine Pragmatik, auf der die Begriffe aufsetzen. Wir benutzen dazu die grundlegenden Konzepte der Mathematik, insbesondere Mengen, Funktionen, Relationen etc. (Wer auch diese Begriffe hinterfragen will, sei auf die entsprechenden Bemühungen der Mathematik verwiesen, wie sie z. B. von Bourbaki [24] ebenso aufwendig wie exzellent präsentiert werden.)

4.1 Items

Die Gesamtheit aller Dinge, die eine Programmiersprache ausmachen – Zahlen, Texte, Listen, Arrays, Funktionen, Typen, Spezifikationen, Module, Bibliotheken usw. – bilden das semantische „Universum“ $\mathbb{U}$ der Sprache (engl.: *universe of discourse*). Für die Elemente dieses Universums brauchen wir einen sehr allgemeinen, generischen Begriff, der so unspezifisch ist wie die Dinge, für die er steht. Wir wählen dazu den Begriff *Item* (der im Englischen genauso nichtssagend ist, wie das deutsche Wort „Ding“, das wir oben verwendet haben).

Definition (Universum, Item)

Die Gesamtheit aller Dinge, die eine Programmiersprache ausmachen, bilden das ***Universum*** $\mathbb{U}$ der Sprache. Die Elemente von $\mathbb{U}$ nennen wir ***Items***.

Die naheliegende Frage ist jetzt: *Wie lässt sich das semantische Universum $\mathbb{U}$ präzise definieren?* Die vorläufige Antwort ist: Wir werden es im Laufe des Buches Stück für Stück erarbeiten.

Als ***Pragmatik*** legen wir hier vorläufig nur fest, dass $\mathbb{U}$ einige vordefinierte Mengen einschließt:

- *Wahrheitswerte* (*Bool*).
- *Zahlen*, insbesondere die natürlichen Zahlen (*Nat*), die ganzen Zahlen (*Int*) und die Gleitpunktzahlen (*Real*); dabei lassen wir offen, wie diese Zahlen genau aussehen (32 Bit, 64 Bit, unbeschränkt).
- *Zeichen* (*Char*); dabei lassen wir offen, ob wir uns auf das ASCII-Alphabet beschränken oder das UNICODE-Alphabet wählen (wobei in beiden Fällen weitere Diversifizierungen möglich sind).
- *Texte* (*String*), also Folgen von Zeichen.

Ebenfalls als Teil der Pragmatik setzen wir die üblichen Operationen auf den Elementen dieser Mengen voraus, also Addition, Multiplikation, Konkatenation etc.

4.2 Das allgemeine Konzept der Gruppen

Zur Motivation betrachten wir drei kleine Beispiele (in Ad-hoc-Notation), auf die wir im Laufe des Kapitels immer wieder zurückgreifen werden.

(1) Punkte im $\mathbb{R}^2$ können in der so genannten analytischen Darstellung mit Betrag und Winkel definiert werden. Und Geraden lassen sich durch zwei Punkte repräsentieren. Das ist in Programm 4.1 illustriert. Dabei nehmen wir an, dass geeignete Typen *Dist* für die nicht-negativen reellen Zahlen und *Angle* für die Winkel zwischen $0°$ und $360°$ verfügbar sind.

Programm 4.1 Eine Linie, gegeben durch zwei Punkte

```
l: Line = { p1: Point =  { dist: Dist = 2.7, angle: Angle = 45 },
            p2: Point =  { dist: Dist = 1.3, angle: Angle = 33 } }
```

(2) Zur Modularisierung von Programmen verwenden wir *Strukturen*. Eine Struktur, die die Typen und Operationen für Punkte im $\mathbb{R}^2$ zusammenfasst, könnte z. B. aussehen wie in Programm 4.2 (wiederum in Ad-hoc-Notation): Hier wird ein Typ *Point* mit den entsprechenden Komponententypen *Dist*

Programm 4.2 Eine Struktur für Punkte im $\mathbb{R}^2$

```
STRUCTURE Point = {
  TYPE Point = { dist: Type = Dist, angle: Type = Angle }
  FUN x: (Point → Real) = λ p • p.dist * cos(p.angle)
  FUN y: (Point → Real) = λ p • p.dist * sin(p.angle)
}
```

und *Angle* sowie ihren Selektoren eingeführt. Außerdem werden zwei Funktionen x und y definiert, die die beiden kartesischen Koordinaten liefern. Die Bedeutung des Identifiers *Type* werden wir in Kapitel 9 kennen lernen.

(3) Wenn wir uns auf das Niveau des Software-Engineerings großer Programmpakete begeben, müssen wir eine Modularisierung in Bibliotheken und Packages vornehmen. Das wird in Programm 4.3 angedeutet. Hier werden einige logisch zusammengehörige Strukturen in einem gemeinsamen *Package* definiert.

Programm 4.3 Modularisierung mittels Packages

```
PACKAGE Geometry = {
  STRUCTURE Point = { ... }
  STRUCTURE Line = { ... }
  STRUCTURE Polygon = { ... }
  STRUCTURE Circle = { ... }
  ...
}
```

Die Ähnlichkeit der obigen Beispiele legt es nahe, das gemeinsame, allen zugrunde liegende Konzept herauszuarbeiten. Bei der Schreibweise können wir eine minimalistische Form wählen, wie sie z. B. von HASKELL angestrebt wird,

oder eine mehr Schlüsselwort-orientierte Form, wie sie z. B. in OPAL bevorzugt wird.

Definition (Gruppe)

Eine ***Gruppe*** g besteht aus n Komponenten, genannt ***Items***, die mit den ***Selektoren*** $a_1, \ldots, a_n$ angesprochen werden. Die Komponenten haben die Typen $T_1, \ldots, T_n$ und ihre Werte sind durch $E_1, \ldots, E_n$ definiert. Auch die Gruppe selbst hat einen Typ T_0.

HASKELL-*Stil*

$$
\begin{array}{l}
g\colon T_0 = \{ \\
\quad a_1\colon T_1 = E_1 \\
\quad \vdots \\
\quad a_n\colon T_n = E_n \\
\}
\end{array}
$$

OPAL-*Stil*

$$
\begin{array}{l}
\text{GROUP } g\colon T_0 = \{ \\
\quad \text{ITEM } a_1\colon T_1 = E_1 \\
\quad \vdots \\
\quad \text{ITEM } a_n\colon T_n = E_n \\
\}
\end{array}
$$

Die einzelnen Komponenten – also einige der E_i – können wieder Gruppen sein.

Die Begriffe „Typ“ und „Wert“ müssen hier in einem erweiterten Sinn verstanden werden. Das werden wir in den Kapiteln 6 bis 9 noch ausgiebig diskutieren. Für den Augenblick wollen wir intuitiv akzeptieren, dass es einen solchen erweiterten Typ- und Wertbegriff gibt.

Anmerkung: Als konsequentere Variante des Schlüsselwort-orientierten Stils könnte man sogar die Klammern $\{..\}$ *weglassen und das Ende der Gruppe durch* ENDGROUP *kennzeichnen.*

4.2.1 *Syntactic sugar*: Schlüsselwörter

Der minimalistische Stil von HASKELL sieht bei kleinen Programmen elegant aus, führt bei großen Programmen aber schnell zu schlechter Lesbarkeit. Außerdem lässt er sich nur bei wenigen Konzepten durchhalten; von einem bestimmten Punkt an werden Schlüsselwörter benötigt, um Mehrdeutigkeiten zu vermeiden. Deshalb werden wir hier von Anfang an etwas spendabler mit ihnen umgehen.

Schlüsselwörter verlangen etwas mehr Schreibarbeit, kompensieren das aber durch eine ganzen Reihe von Vorteilen. Vor allem lassen sich Mehrdeutigkeiten vermeiden, was nicht nur für den Compiler hilfreich ist, sondern – wichtiger noch – für den menschlichen Leser. Außerdem wird die Behandlung von Tippfehlern wesentlich erleichtert, weil der Compiler viel schneller wieder in Tritt kommt (ebenso wie der menschliche Leser).

Damit erhalten wir aber das Problem, beim Sprachdesign geeignete Schlüsselwörter wählen zu müssen. Dabei ist die Gefahr der oben erwähnten *Featuritis* besonders groß. Unsere grundlegende Herangehensweise an Sprachkonzepte führt uns sehr schnell auch zu einem neuen Ansatz für Schlüsselwörter. Wir brauchen – ähnlich wie bei Typen – eine Art von *Vererbungshierarchie* für Schlüsselwörter. Jedes Schlüsselwort hat eine semantische Bedeutung und hilft somit sowohl dem Compiler als auch dem menschlichen Leser.

Das hat sich schon in der obigen Definition gezeigt. Denn einige der Items a_i können wiederum Gruppen sein, was eigentlich das Schlüsselwort GROUP anstelle von ITEM erfordern würde.

Festlegung (Hierarchie von Schlüsselwörtern)

Die *Schlüsselwörter* unserer Sprache sind *hierarchisch* organisiert. An der Spitze steht das völlig unspezifische Schlüsselwort ITEM. Wenn es sich bei dem Item um eine Gruppe handelt, darf das Schlüsselwort GROUP benutzt werden.

Weitere Spezialisierungen werden wir im Laufe der Zeit kennen lernen: VAL, FUN, TYPE, STRUCTURE, PACKAGE etc. Aber es ist immer zulässig, anstelle eines spezifischen ein schwächeres Schlüsselwort zu verwenden, im Extremfall GROUP oder sogar nur ITEM.

Wenn wir über allgemeine Konzepte sprechen, dann verwenden wir meistens die unspezifischen Schlüsselwörter wie ITEM oder GROUP (und manchmal werden wir uns auch die Freiheit nehmen, in den Schlüsselwort-freien HASKELL-Stil zu verfallen). Aber es ist klar, dass die jeweiligen Aussagen auch für die Spezialisierungen wie FUN, TYPE, PACKAGE etc. gelten.

Anmerkung 1: Die Hierarchie ist nicht einfach. Zum Beispiel können Typen manchmal Gruppen sein, manchmal aber auch nicht. Deshalb kann TYPE *manchmal durch* GROUP *ersetzt werden, manchmal aber auch nur durch* ITEM*.*

Anmerkung 2: Da alle Schlüsselwörter letztlich nur Spezialisierungen von ITEM *sind, erhalten wir prinzipiell die Möglichkeit, unsere Programmiersprache stark zu flexibilisieren. Denn es gibt keinen Grund, weshalb nicht einzelne Programmierer neue Schlüsselwörter einführen sollten. (Dies wäre ein ähnlicher Mechanismus wie die „Stereotypes“ in* UML*; wir werden diese Möglichkeit hier aber nicht systematisch ausarbeiten, sondern nur punktuell andeuten.)*

Anmerkung 3: Manchmal stoßen wir auf einen Konflikt zwischen konzeptuell sauberen und traditionell geläufigen Notationen. In diesen Fällen können wir spezielle Schlüsselwörter benutzen, um einen Kompromiss zwischen beiden Wünschen zu erlauben.

4.2.2 Selektoren und die Semantik von Gruppen

Es gibt zwei grundsätzliche Varianten für die Interpretation von Gruppen und ihren Selektoren. Wir zeigen zunächst die Variante, die wir tatsächlich benutzen; die andere werden wir im Anschluss nur kurz skizzieren.

Definition (Selektor)

In einer Gruppe der Art

GROUP g: $T_0 = \{$
  ITEM a_1: $T_1 = E_1$
  $\vdots$
  ITEM a_n: $T_n = E_n$
$\}$

sind die ***Selektoren*** $a_1, \ldots, a_n$ spezielle Funktionen, die definiert sind als

a_i: $\{ g \} \to T_i$
$a_i(g) = E_i$ bzw. $g\,.\,a_i = E_i$ $--$ *Funktions- bzw. Selektorschreibweise*

Das heißt, der Definitionsbereich ist die einelementige Menge $\{g\}$ und der Wertebereich ist der angegebene Typ T_i. Das semantische Objekt g wird dabei *coalgebraisch* definiert [49] als dasjenige Objekt, das implizit durch die *Beobachtungsfunktionen* $a_1, \ldots, a_n$ charakterisiert wird. Aus Gründen der besseren Lesbarkeit schreiben wir die ***Selektion*** üblicherweise mit Hilfe des „.“-Operators aus Abschnitt 1.2.1.

Das ist eine sehr einfache semantische Konzeption, weil Gruppen nichts anderes sind als Mengen von Funktionen, also wohl definierte und einfache mathematische Konzepte. Wir werden im weiteren Verlauf dieses Kapitels sehen, dass damit selbst große modularisierte Softwaresysteme eine klare und einfache semantische Fundierung erhalten und dass sich viele Techniken des Software-Engineerings klar und sauber definieren lassen.

Weil die Funktionsschreibweise $a_i(g)$ sehr unhandlich ist, vor allem bei Selektionsketten der Bauart $radius(Circle(Geometry(Mathematics)))$, verbinden wir sie meistens mit dem „.“-Operator aus Abschnitt 1.2.1. Zur Erinnerung, dieser Operator ist als polymorphe Funktion definiert:

FUN $\_\,.\,\_$: $\alpha \times (\alpha \to \beta) \to \beta$
DEF $x.f = f(x)$

Damit lassen sich Selektionen in der traditionellen Form $g\,.\,a_i$ schreiben, also z. B. $Mathematics\,.\,Geometry\,.\,Circle\,.\,radius$. Man beachte aber, dass damit *kein* zusätzliches Konzept eingeführt wird; der „.“-Operator ist eine normale polymorphe Funktion höherer Ordnung.

Anmerkung 1: Wir können hier nicht detaillierter auf die Methodik der coalgebraischen Definition eingehen, merken aber an, dass wir in dieser Sichtweise einen charmanten Nebeneffekt erhalten. Wenn compilerintern zu einer Gruppe weitere Attribute hinzugefügt werden, verletzt das nicht ihre coalgebraische Semantik.

Anmerkung 2: Es hätte auch eine alternative Interpretation gegeben, die eher den Traditionen älterer Programmiersprachen wie COBOL *oder* PASCAL *entspricht und die implizit auch in* ML *oder* HASKELL *enthalten ist. Bei dieser alternativen Sicht fasst man die Gruppe g als Funktion auf:*

$g\colon \{ a_1, \ldots, a_n \} \to T_1 \oplus \ldots \oplus T_n$ *-- alternative Möglichkeit*
$g(a_1) = E_1 \quad \ldots \quad g(a_n) = E_n$

Der Definitionsbereich dieser Funktion ist die Menge der Selektoren $a_1, \ldots, a_n$, die dabei als neue primitive Werte eingeführt werden (und keine andere Eigenschaft haben als die, verschieden zu sein). Der Wertebereich ist die „Summe" der Typen $T_1, \ldots, T_n$. Das ist aber gerade das Problem dieser Sichtweise: Es ist nicht ohne weiteres klar, wie der Begriff „Summe" sauber definiert werden kann, da die T_i – wie wir in Kapitel 6 bis 9 noch sehen werden – sehr komplexe Gebilde sein können.

4.3 Environments und Namensräume

Im Folgenden soll die Idee der Gruppen weiter ausgebaut werden. Zu diesem Zweck erweisen sich Begriffe aus der Welt des Compilerbaus und der Sprachsemantik als hilfreich (was nicht überraschen sollte). Die zentrale Frage, die wir beantworten müssen, lautet: *Welche Identifier sind wo bekannt und wie dürfen sie verwendet werden?*

Zur Illustration der folgenden Überlegungen betrachten wir das schematische Beispiel in Programm 4.4. Wir wollen hier über vollständige Programme

Programm 4.4 Ein schematisches Beispielprogramm

```
𝕎 = {                                                    -- the "World"
  PACKAGE P = {
    STRUCTURE A = { ITEM a = ... a ... B.b ... Q.S.a ...}
    STRUCTURE B = { ITEM b = ... b ... A.a ... Q.S.a ...}
  }
  PACKAGE Q = {
    STRUCTURE S = { ITEM a = ... a ... P.A.a ... P.B.b ...}
  }
}
```

reden (einschließlich aller Bibliotheken etc.), die letztlich eine in sich abgeschlossene „Welt" darstellen. Deshalb verwenden wir den Identifier $\mathbb{W}$ zur Charakterisierung des Gesamtprogramms. (Was genau ein „Programm" ist, werden wir in Abschnitt 4.7.1 noch genau klären; im Augenblick beschränken wir uns auf die Feststellung, dass es im Wesentlichen eine Gruppe ist, und zwar die äußerste.)

In Tabelle 4.1 sind alle Items aus Programm 4.4 zusammen mit ihren *Namen* aufgeführt.

Dabei benutzen wir den Begriff *Namen* für die eindeutige Kennzeichnung der Items. Und das erfordert im Allgemeinen die Angabe der gesamten Selektorkette. Die Selektoren selbst sind dabei *Identifier* im üblichen Sinn von Programmiersprachen, also meistens Zeichenfolgen, die aus Buchstaben und

Item	Name	
	als Funktion	als Selektorkette
≪world≫	$\mathbb{W}$	$\mathbb{W}$
≪package P≫	$P(\mathbb{W})$	$\mathbb{W}.P$
≪package Q≫	$Q(\mathbb{W})$	$\mathbb{W}.Q$
≪structure A≫	$A(P(\mathbb{W}))$	$\mathbb{W}.P.A$
≪structure B≫	$B(P(\mathbb{W}))$	$\mathbb{W}.P.B$
≪structure S≫	$S(Q(\mathbb{W}))$	$\mathbb{W}.Q.S$
≪item a≫	$a(A(P(\mathbb{W})))$	$\mathbb{W}.P.A.a$
≪item b≫	$b(B(P(\mathbb{W})))$	$\mathbb{W}.P.B.b$
≪item a≫	$a(S(Q(\mathbb{W})))$	$\mathbb{W}.Q.S.a$

Tab. 4.1: Die Items aus Programm 4.4 und ihre Namen

Ziffern zusammengesetzt sind; wir betrachten aber auch Grapheme wie „ + “, „ → “ etc. als Identifier.

Wie man sieht, werden mehrfach vorkommende Identifier wie a durch ihre Erweiterung zu vollständigen Namen eindeutig gemacht – und zwar eindeutig im gesamten Programm. Aus Gründen der leichten Benutzbarkeit und Lesbarkeit muss es natürlich möglich sein, dass der Programmierer nur mit den kurzen Identifieren arbeitet und der Compiler die Erweiterung zu den vollen Namen automatisch vollzieht.[1] Darauf kommen wir gleich noch zurück.

Definition (Name)

Ein ***Name*** ist ein Funktionsausdruck der Art $A(P(\mathbb{W}))$ oder $a(A(\_))$, der ein Item liefert oder eine Funktion, die bei Anwendung auf eine geeignete Gruppe ein Item liefert. Im ersten Fall sprechen wir von einem ***vollständigen Namen***, im zweiten von einem ***partiellen Namen***.

Wir schreiben Namen meistens als *Selektorketten* der Art $\mathbb{W}.P.A$ oder $A.a$; allerdings sind dann partielle und vollständige Namen nicht mehr gut unterscheidbar.

Eine Menge von Namen bezeichnen wir auch als ***Namensraum***.

Anmerkung: Im Zusammenhang mit Overloading (s. Abschnitt 4.4) muss man das Konzept der Namen etwas erweitern, indem die Namen mit Typen annotiert werden. Das grundsätzliche Prinzip bleibt aber erhalten.

[1] Wie wichtig das ist, sieht man bei den Compilern für C. Hier gibt es diesen Komfort nicht, was vielen Generationen von Programmierern das Leben schwer gemacht und die Industrie viel Geld gekostet hat (und immer noch kostet).

4.3.1 Environments

Wenn man ein (syntaktisch wohl definiertes) Programmfragment p betrachtet wie z. B. eine Funktion oder eine Gruppe, dann zerfällt das Gesamtprogramm in zwei Teile (vgl. Abbildung 4.1), und zwar in

- das betrachtete Fragment p selbst und
- den ***Kontext*** (also das „Restprogramm“).

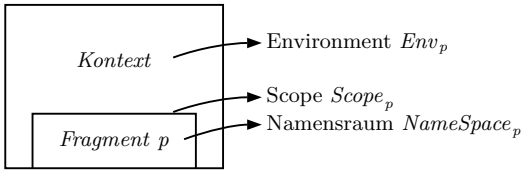


Abb. 4.1: Fragment und Kontext

Der Kontext des Fragments p induziert einen speziellen Namensraum, das so genannte *Environment* Env_p.

Definition (Environment)

Jedes syntaktisch wohl definierte Programmfragment p besitzt ein ***Environment*** Env_p. Dieses Environment ist die Menge aller Namen, die im Kontext von p definiert werden (und somit in p benutzbar sind).

Man beachte, dass der Kontext im Allgemeinen riesig ist, weil er unter anderem alle verwendeten Bibliotheken mit umfasst. Deshalb werden im Compilerbau effiziente Verfahren zur Repräsentation und Speicherung der Environments verwendet.

4.3.2 Scopes und lokale Namensräume

Einige syntaktische Konstrukte – z. B. Gruppen und Funktionen – definieren lokale Namen. Im Beispiel des schematischen Programms 4.4 definiert z. B. das Package P die Namen

$$\{\, \mathbb{W}.P.A,\ \mathbb{W}.P.A.a,\ \mathbb{W}.P.B,\ \mathbb{W}.P.B.b \,\}$$

Diese Menge stellt den *lokalen Namensraum* von P dar. Man beachte, dass es sich dabei um die vollen Namen handelt.

Definition (lokaler Namensraum)

Gewisse syntaktische Konstrukte – z. B. Gruppen – definieren neue Namen. Die Menge der lokalen Namen, die in einem solchen Programmfragment p

eingeführt werden, bezeichnen wir als ***lokalen Namensraum*** $NameSpace_p$. Den zugehörigen Programmbereich bezeichnen wir als ***Scope*** $Scope_p$.

Im Fall von Gruppen lassen sich der Scope und der lokale Namensraum durch den Namen der Gruppe identifizieren. Im Programm 4.4 induziert das Package P den Scope $\mathbb{W}.P$; der zugehörige Namensraum ergibt sich, indem die Selektoren der Items von P an den Scopenamen angehängt werden.

Anmerkung: Bei Funktionen, LET*-Ausdrücken etc. ist die Situation analog; allerdings müssen wir manchmal, z. B. bei* LET*-Ausdrücken, anonyme Scopenamen benutzen, was wir einfach durch Nummerierung darstellen können. Betrachten wir das artifizielle Programm*

$$\text{FUN } f = \lambda x, y \bullet (\text{LET } a = x * x \text{ IN } a + 1) * (\text{LET } b = y * y \text{ IN } b + 1)$$

Wenn der umfassende Scope σ ist, dann haben wir hier die Namen

$$\sigma.f,\ \sigma.f.x,\ \sigma.f.y,\ \sigma.f.1.a,\ \sigma.f.2.b.$$

4.3.3 Namenserkennung im Scope

Wenn wir immer mit den vollen Namen arbeiten würden, hätten wir sofort ein Problem mit den unlesbar langen Selektorketten. Das deutet sich schon in dem kleinen Programm 4.4 und der zugehörigen Tabelle 4.1 an. Das Programm zeigt auch schon das Lösungsprinzip. *Innerhalb ihres Scopes müssen Namen nur partiell angegeben werden.* Zur Illustration wiederholen wir in Programm 4.5 das Programm 4.4, wobei jetzt aber alle angewandten Namen voll ausgeschrieben werden. Man sieht sofort: diese Notation ist völlig unlesbar.

Programm 4.5 Das Beispielprogramm 4.4 mit vollen Namen

$\mathbb{W} = \{$
  PACKAGE $P = \{$
    STRUCTURE $A = \{$ ITEM $a = \ldots \mathbb{W}.P.A.a \ldots \mathbb{W}.P.B.b \ldots \mathbb{W}.Q.S.a \ldots\}$
    STRUCTURE $B = \{$ ITEM $b = \ldots \mathbb{W}.P.B.b \ldots \mathbb{W}.P.A.a \ldots \mathbb{W}.Q.S.a \ldots\}$
  $\}$
  PACKAGE $Q = \{$
    STRUCTURE $S = \{$ ITEM $a = \ldots \mathbb{W}.Q.S.a \ldots \mathbb{W}.P.A.a \ldots \mathbb{W}.P.B.b \ldots\}$
  $\}$
$\}$

Außerdem hätte das Einschieben einer neuen Modularisierungsebene – was aus softwaretechnischen Gründen jederzeit möglich sein muss – katastrophale Folgen.

Die Lösung des Problems ist wohl bekannt und wurde im ursprünglichen Programm 4.4 auch realisiert: Innerhalb des eigenen Scopes reicht der partielle Name, denn er wird automatisch zum vollen Namen erweitert.

Formal können wir den Prozess der Namensauflösung einfach beschreiben. Wir tun dies zunächst in der Funktionsschreibweise. Um festzustellen, ob ein gegebener Identifier ein Selektor sein könnte, ergänzen wir ihn formal mittels des Wildcard-Symbols zu einer Funktion. Im Originalprogramm 4.4 hätten wir damit z. B. die Zeile

$$\text{STRUCTURE } A = \{\,\text{ITEM } a = \ldots a(\_)\ldots b(B(\_))\ldots a(S(Q(\_)))\ldots\}$$

Die rechte Seite der Deklaration von a hat als Environment alle Namen in Tabelle 4.1. Gegen diese Namen müssen wir die Funktionen *matchen*, was – wenn wir keine weiteren Einschränkungen vornehmen – folgende Paarungen liefert:

$$\begin{array}{lcl} a(\_) & \leftrightarrow & a(A(P(\mathbb{W}))),\ a(S(Q(\mathbb{W}))) \\ b(B(\_)) & \leftrightarrow & b(B(P(\mathbb{W}))) \\ a(S(Q(\_))) & \leftrightarrow & a(S(Q(\mathbb{W}))) \end{array}$$

Wenn bei diesem Matching mehr als ein Treffer erzielt wird, ist der partielle Name mehrdeutig, was zu einer Fehlermeldung führt.[2]

In unserem obigen Beispiel erwarten wir eigentlich, dass *keine* Fehlermeldung kommt; denn es ist klar, dass mit dem $a(\_)$ der Selektor von A gemeint ist und nicht der von S. Deshalb wird der Matching-Prozess noch um eine Bedingung erweitert: Die Ergänzungen zu vollen Namen müssen zum aktuellen Scope „passen“. In unserem Beispiel ist der aktuelle Scope die Struktur $A(P(\mathbb{W}))$, was bedeutet, dass folgende drei Namen auf dem Pfad liegen: $\{A(P(\mathbb{W})),\ P(\mathbb{W}),\ \mathbb{W}\}$. Nur diese dürfen als Argumente herangezogen werden, bevor das Matching gegen Tabelle 4.1 erfolgt. Damit wird die Auswahl eindeutig:

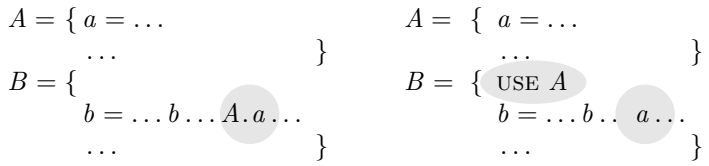
$$\begin{array}{lcll} A(P(\mathbb{W})),\ P(\mathbb{W}),\ \mathbb{W} & \vdash & a(\_) & \leftrightarrow\ a(A(P(\mathbb{W}))) \\ A(P(\mathbb{W})),\ P(\mathbb{W}),\ \mathbb{W} & \vdash & b(B(\_)) & \leftrightarrow\ b(B(P(\mathbb{W}))) \\ A(P(\mathbb{W})),\ P(\mathbb{W}),\ \mathbb{W} & \vdash & a(S(Q(\_))) & \leftrightarrow\ a(S(Q(\mathbb{W}))) \end{array}$$

Übrigens: Wenn wir das Ganze auf der Basis der Selektorschreibweise betrachten, dann muss z. B. dem partiellen Selektorpfad $\_.B.b$ der vollständige Pfad $\mathbb{W}.P.B.b$ zugeordnet werden. Das heißt, wir müssen volle Selektorpfade finden, die mit dem gegebenen partiellen Selektorpfad enden.

4.3.4 Namenserkennung außerhalb des Scopes (USE)

Aus pragmatischen Gründen möchte man manchmal auch partielle Namen verwenden, die *nicht* zum aktuellen Scope passen. Die Lösung dieses Problems ist wohl bekannt, spätestens seit dem `with`-Konstrukt von PASCAL oder dem `open`-Konstrukt von ML; in HASKELL wird dafür das Schlüsselwort `import` benutzt. Wir verwenden das Schlüsselwort USE, um den Matching-Prozess entsprechend zu erweitern. Die folgenden beiden Programme sind äquivalent.

[2] Im Allgemeinen nimmt man noch Typinformationen hinzu, um die Zahl der Mehrdeutigkeiten zu reduzieren.



Das Prinzip funktioniert unverändert über mehrere Stufen hinweg. Das sieht man im Programm 4.6, das wieder unser schematisches Beispiel aufgreift, jetzt aber mit entsprechenden USE-Klauseln.

Programm 4.6 Das schematische Beispielprogramm mit USE-Klauseln

```
𝕎 = {
  PACKAGE P = {
    STRUCTURE A = { USE B
                    ITEM a = ... a ... b ... Q.S.a ...}
    STRUCTURE B = { USE A
                    ITEM b = ... b ... a ... Q.S.a ...}
  }
  PACKAGE Q = {
    USE P
    STRUCTURE S = { USE B
                    ITEM a = ... a ... A.a ... b ...}
  }
}
```

Der Prozess lässt sich mit unseren Definitionen sehr leicht erklären. Betrachten wir z. B. das USE B in der Struktur A. Der partielle Name $B(\_)$ selbst wird nach unseren obigen Regeln eindeutig zu $B(P(\mathbb{W}))$ erweitert. Damit wird die Menge der Elemente zur weiteren Auflösung von drei auf vier Elemente erweitert: $\{A(P(\mathbb{W})), B(P(\mathbb{W})), P(\mathbb{W}), \mathbb{W}\}$. (Eigentlich generiert $B(P(\mathbb{W}))$ drei Namen, aber zwei davon sind bei $A(P(\mathbb{W}))$ schon da, so dass bei der Mengenvereinigung nur vier Elemente bleiben.) Bezogen auf diese Menge wird jetzt auch $b(\_)$ als $b(B(P(\mathbb{W})))$ erkannt.

Man rechnet schnell nach, dass „verschränkte Rekursion“, bei der in A ein USE B erfolgt und in B ein USE A, völlig problemlos ist. Es werden ja jeweils nur einige Elemente für das Matching bereitgestellt.

Übrigens: Wenn wir in S auch noch USE A schreiben würden, um im Rumpf $A.a$ zu a abzukürzen, hätten wir eine Mehrdeutigkeit erzeugt, weil dann sowohl $A(P(\mathbb{W}))$ als auch $S(Q(\mathbb{W}))$ in der Matching-Menge liegen würden.

Wenn man sich die Importe z. B. bei der Sprache JAVA ansieht, dann sollte man noch ein paar *pragmatische Spezialnotationen* einführen. Man könnte z. B. in unserem obigen Beispiel im Package Q sagen USE $P.*$. Das wäre dann äquivalent zu den beiden Termen USE $P.A$ und USE $P.B$. Auch iterative Wildcards sind möglich, also $P.*.*$ usw. Eine nette Variante (die sich in

JAVA nicht findet) wäre auch $P.**$, womit das gesamte Package P in seiner ganzen Tiefe bereitgestellt würde.

Definition (Scope-Erweiterung, USE)

Mit der Notation USE s wird der Namensraum $NameSpace_s$ zum aktuellen Scope hinzugefügt und damit bei der Auflösung partieller Namen ebenfalls herangezogen.

Als notationelle Erleichterung können auch Wildcards verwendet werden wie z. B. $P.*$, $P.*.*$, $P.**$ etc.

Man beachte: Die USE-Notation ist nur eine *Schreibabkürzung*. Sie hat keine weiteren semantischen Konsequenzen. Diese werden erst mit anderen Mitteln erreicht, die wir in Kapitel 5 unter den Stichwörtern *Vererbung* und *Import/Export* diskutieren werden.

4.4 Overloading

Dass die gleichen Namen in verschiedenen Scopes definiert werden dürfen, ist eine große Hilfe für das Schreiben lesbarer Programme. Aber es gibt auch Situationen, in denen man gerne die gleichen Namen mehrfach im gleichen Scope einführen würde. Ein typisches Beispiel ist etwa die Rotation einer geometrischen Figur im $\mathbb{R}^2$:

FUN *rotate*: *Shape* → *Shape*
FUN *rotate*: *Shape* → *Point* → *Shape*

Im ersten Fall erfolgt die Rotation um den Nullpunkt, im zweiten Fall um einen als Parameter angegebenen Punkt. Es wäre offensichtlich kontraproduktiv, wenn der Programmierer sich hier zwei verschiedene Namen ausdenken müsste. Da sowohl Menschen als auch Compiler aus dem Kontext jeweils ablesen können, welche der beiden Funktionen gemeint ist, gibt es auch keinen Grund, hier verschiedene Namen zu erzwingen.

Definition (Overloading)

Wenn ein Name (im gleichen Scope) mehrfach eingeführt wird, spricht man von ***Überlagerung*** oder ***Overloading***; dabei müssen die beiden Namen unterschiedliche Typen haben.

In der Literatur gibt es zwei primäre Versionen von Overloading. In den meisten Sprachen (sofern sie überhaupt Overloading haben) wird gefordert, dass sich die *Parametertypen* unterscheiden müssen. Ein bekanntes Beispiel einer solchen Sprache ist JAVA. In anderen Sprachen wird etwas mehr Komfort geboten: Es müssen sich *Parameter- oder Resultattypen* unterscheiden, wie es z. B. in OPAL der Fall ist.

4.5 Beispiele für Strukturen und Packages

Unser flexibler Umgang mit Schlüsselwörtern ermöglicht eine klare softwaretechnische Strukturierung, ohne neue semantische oder sonstige Konzepte einführen zu müssen. Wir führen einfach drei neue Schlüsselwörter als Synonyme für GROUP ein, für die wir allerdings gewisse Rahmenbedingungen fordern, um den Dokumentationswert zu erhöhen. Die drei Schlüsselwörter stehen in folgender Hierarchie:

1. Auf der obersten Stufe steht LIBRARY.
2. Auf der zweiten Stufe steht PACKAGE.
3. Auf der dritten Stufe steht STRUCTURE.

Das heißt z. B., dass Packages in Librarys enthalten sein können, aber nicht umgekehrt. Ansonsten können wir aber liberal sein. So kann man zulassen, dass Strukturen auch direkt in Librarys enthalten sein dürfen, ohne dass man zwingend noch ein Package als Rahmen herumbaut.

Wir können die Relation auch reflexiv gestalten, so dass Librarys Sublibrarys enthalten können, Packages Subpackages und Strukturen Substrukturen.

Da alle diese Schlüsselwörter nur Synonyme für GROUP sind, können derartige Regelungen rein nach dokumentatorischen Gesichtspunkten gestaltet werden, ohne semantischen Aufwand zu verursachen.

Die wichtigsten Ebenen für die softwaretechnische Modularisierung sind Strukturen und Packages. Wir erläutern ihre Rolle anhand von typischen Beispielen.

Definition (Struktur)

Eine *Struktur* ist eine spezielle Gruppe, die gewissen Randbedingungen genügt (s. Abschnitt 9.2).

Eine wichtige Gruppe von Standardstrukturen gibt es – in mehr oder weniger reichhaltiger Form – in den meisten Programmiersprachen: die Zahlen. Ein entsprechendes Package könnte dann etwa aussehen wie in Programm 4.7.

Dem Programmierer sollten prinzipiell die natürlichen Zahlen $\mathbb{N}$, die ganzen Zahlen $\mathbb{Z}$, die rationalen Zahlen $\mathbb{Q}$, die relleen Zahlen $\mathbb{R}$ und die komplexen Zahlen $\mathbb{C}$ zur Verfügung stehen. Dabei muss man pragmatisch unterscheiden zwischen den schnellen Varianten der Maschinenzahlen, also z. B. *Int*, *Long*, *Float* oder *Double*, und den langsamen Varianten der selbstdefinierten Zahlen. Zu letzteren gehören z. B.

- *BigInt*: unbeschränkt große Zahlen, die als Listen von *Int*-Werten repräsentiert werden; diese Werte fungieren als „Ziffern“ in einem Stellenwertsystem zur Basis $B = 2^{32}$.
- *Rat*: rationale Zahlen, die als Paare von (unbeschränkten) ganzen Zahlen repräsentiert werden, wobei im Allgemeinen noch Normierungen wie Teilerfremdheit und positive Nenner sichergestellt werden.

Programm 4.7 Das Package der Zahl-Strukturen

```
PACKAGE Arithmetic = {
  STRUCTURE Natural      = { TYPE Nat
                             FUN _ + _: Nat × Nat → Nat
                             ...                          }
  STRUCTURE Integer      = { TYPE Int ... }
  STRUCTURE LongInteger  = { TYPE Long ... }
  STRUCTURE BigInteger   = { TYPE BigInt ... }
  STRUCTURE Rational     = { TYPE Rat ... }
  STRUCTURE Float        = { TYPE Real ... }
  STRUCTURE Double       = { TYPE Double ... }
  STRUCTURE Complex      = { TYPE Complex ... }
}
```

- *Complex*: komplexe Zahlen, die als Paare von reellen Zahlen dargestellt werden, wobei man noch zwischen *Float* und *Double* unterscheiden kann.

Für alle diese Strukturen werden die üblichen arithmetischen Operationen bereitgestellt. (In Kapitel 9 werden wir Techniken kennenlernen, mit denen sich die Gemeinsamkeiten dieser Strukturen explizit charakterisieren lassen.) Dabei empfiehlt es sich, im Interesse der Anwender von vornherein eine „große“ Lösung zu entwerfen, und nicht wie bei JAVA die Operationen auf verschiedene Packages zu verteilen, aus denen sie vom Benutzer mühsam zusammengestellt werden müssen.

Programm 4.8 Das Package der Geometrie-Strukturen

```
PACKAGE Geometry = {
  STRUCTURE Point      = { TYPE Point ... }
  STRUCTURE Line       = { TYPE Line ... }
  STRUCTURE Triangle   = { TYPE Trianlge ... }
  STRUCTURE Rectangle  = { TYPE Rect ... }
  STRUCTURE Polygon    = { TYPE Poly ... }
  STRUCTURE Oval       = { TYPE Oval ... }
  STRUCTURE Circle     = { TYPE Circle ... }
  ...
}
```

Analog zu den zahlartigen Strukturen kann man z. B. auch geometrische Strukturen zu einem Package zusammenfassen. Programm 4.8 skizziert den Aufbau eines entsprechenden Packages *Geometry*, in dem Punkte, Linien, Dreicke, Polygone, Kreise etc. definiert werden. Dabei sollten sowohl Berechnungsalgorithmen, also Analytische Geometrie, als auch Graphikalgorithmen,

also Darstellende Geometrie, enthalten sein. Außerdem wird man das Package in zwei Versionen brauchen, nämlich als *Geometry2D* und *Geometry3D*.

Ähnliche Packages lassen sich für diverse Varianten der Numerischen Mathematik aufbauen, z. B. für Lineare Algebra, Differentialgleichungen etc.

Dann kann man im strukturellen Aufbau noch einen Schritt weiter gehen und ein übergeordnetes Package einführen, das die genannten Packages enthält. Das ist in Programm 4.9 skizziert.

Programm 4.9 Das Package der Mathematik-Packages

```
PACKAGE Mathematics = {
  PACKAGE Arithmetic  = { ... }
  PACKAGE Geometry    = { ... }
  PACKAGE Numerics    = { ... }
  ...
}
```

Aus den Bibliotheken der moderneren Programmiersprachen kennt man noch viele weitere derartige Standardpackages. Typische Beispiele wären etwa noch

```
PACKAGE Gui = {...}
PACKAGE Text = {...}
```

Hier ist nicht der Platz, um diese Strukturierung im Detail zu entwerfen; solche softwaretechnischen Designfragen sind auch nicht Gegenstand dieses Buches. Für uns ist vor allem die Erkenntnis wichtig, dass dieser ganze Bereich letztlich auf einem einzigen programmiersprachlichen Konzept beruht, nämlich den *Gruppen*. Und die wiederum sind eigentlich nur spezielle Sammlungen von Funktionen.

4.6 Weitere syntaktische Spielereien

Es gibt erstaunlich viele syntaktische Spielereien rund um das elementare Konzept von Gruppen. Im Folgenden wollen wir nur die zwei wichtigsten ansprechen.

4.6.1 Verteilte Definition von Items

In der HASKELL-artigen Notation werden die Einführung der Items, ihre Typisierung und ihre Definition in einem kompakten Konstrukt aufgeschrieben. Das entspricht programmiersprachlichen Traditionen seit ALGOL, die sich auch in C und JAVA erhalten haben. Man kann aber auch die Beschreibung der Items

in einzelne Bestandteile zerlegen (was es übrigens schon in ALGOL gab, aber auch in LISP, FORTRAN und COBOL); das entspricht dann der Vorgehensweise von OPAL. Das folgende schematische Beispiel illustriert die Situation:

```
GROUP G = {
  ITEM f                  -- Einführung des Items f
  FUN f: α → β            -- Typisierung des Items f
  DEF f(0) = ...          -- Definition des Items f (erstes Pattern)
  DEF f(n + 1) = ...      -- Definition des Items f (zweites Pattern)
      ...
}
```

Dies entspricht der kompakten Einführung (vgl. Abschnitt 1.1.8)

```
GROUP G = {
  ITEM f: α → β = λx • IF x MATCHES 0     THEN ...
                      IF x MATCHES n + 1 THEN ... FI
      ...
}
```

Ganz analog kann man z. B. die Einführung des Typs *Point* in verteilter Form oder kompakt vornehmen. (Das Schlüsselwort KIND wird in Abschnitt 6.1.6 genauer erläutert.)

```
GROUP G = {
  ITEM Point
  KIND Point: Type
  DEF Point = { dist = Dist, angle = Angle }
      ...
}
```

Der gleiche Effekt lässt sich auch kompakt erzielen:

```
GROUP G = {
  ITEM Point: Type = { dist = Dist, angle = Angle }
      ...
}
```

Wenn wir das Schlüsselwort TYPE zur Abkürzung verwenden – TYPE *Point* steht dann kurz für ITEM *Point*: *Type* – können wir auch schreiben:

```
GROUP G = {
  TYPE Point = { dist = Dist, angle = Angle }
      ...
}
```

Welche Form man vorzieht, ist weitgehend Geschmackssache. Ein kleines Problem bei der verteilten Beschreibung kann im Zusammenhang mit überlagerten Funktionen entstehen. Weil dabei der gleiche Selektorname mehrfach vergeben wird, kann man ihn nicht ohne irgendeine Form der Typannotation hinschreiben.

4.6.2 Tupel als spezielle Gruppen

Gruppen sind offensichtlich ein ganz elementares Sprachkonstrukt. Trotzdem sind sie in Programmiersprachen nur selten explizit zu finden. (Die *Records* von ML sind eine der wenigen Ausnahmen.) Implizit tauchen sie häufiger auf, meist als Klassen, Module, Packages etc.

Das liegt daran, dass die meisten Sprachen sich auf einen Spezialfall der Gruppen konzentrieren, nämlich ***Tupel***. Zumindest beim Programmieren-im-Kleinen haben Tupel zwei unbestrittene Vorteile: Sie sind notationell knapper und sie sind von der Mathematik her vertrauter. Betrachten wir ein schlichtes Beispiel:

FUN $pow\colon Real \times Nat \to Real$
DEF $pow = \lambda x, n \bullet \ldots x \ldots n \ldots$

Ein Aufruf dieser Funktion hat dann eine Form wie

$\ldots pow(3.7, 3) \ldots$

Compilertechnisch kann man das so auffassen, dass implizit ein anonymer Typ eingeführt wird, so dass folgende Situation entsteht

TYPE $Anonym = \{ x = Real, n = Nat\}$ $--$ *implizit generiert*
FUN $pow\colon Anonym \to Real$ $--$ *implizit adaptiert*
DEF $pow\ \ a = \ldots a.x \ldots a.n \ldots$ $--$ *implizit adaptiert*

Der Aufruf entspricht dann der Einführung einer entsprechenden Gruppe:

$\ldots pow\{x = 3.7, n = 3\} \ldots$

Mit anderen Worten, Tupel sind nichts anderes als Gruppen, bei denen „unsichtbare“ Selektoren erzeugt werden, denen die Items abhängig von der Reihenfolge zugeordnet werden.

4.7 Programme und das Betriebssystem

Wir haben bisher eine rein konzeptuelle Sicht auf unsere Sprachkonstrukte genommen. Aber wenn wir über Dinge wie Strukturen und gar Packages reden, dann haben wir es mit großen Programmsystemen zu tun. Und damit tritt das pragmatische Problem der Speicherung und Verwaltung der entsprechenden Programmteile in Dateien und Directorys auf.

Traditionell haben die Programmiersprachen diesen Bereich des Programmierens-im-ganz-Großen ignoriert. Die Organisation wurde dem Dateisystem überlassen oder speziellen Programmierumgebungen, den so genannten IDEs (engl.: *Integrated Development Environment*). Spätestens mit JAVA und .NET hat sich die Situation aber geändert. Hier wird auch diese Ebene zumindest partiell in die Programmiersprachen einbezogen.

Unser Ansatz geht diesen Schritt konsequent zu Ende. Bei unserer generellen Sichtweise werden alle Ebenen der Programmstrukturierung ganz natürlich

erfasst – und zwar unter einem einzigen semantischen Konzept. Das zeigt, dass die traditionelle Herangehensweise sehr ad hoc war.

Was jetzt noch zu klären bleibt, ist das Verhältnis der Sprachstrukturen zu den Elementen des Betriebssystems, also insbesondere zu Dateien und Directorys.

4.7.1 Was ist eigentlich ein Programm?

Diese Frage klingt trivial, ist aber in der Realität gar nicht so einfach – und sehr oft miserabel gelöst. Man betrachte etwa die schlimme Festlegung in JAVA, wo man in einer seiner Klassen, z. B. der Klasse `MyProg`, das Monstrum „`public static void main(String[] args) { ... }`“ hinschreiben muss, nur damit das System beim Aufruf `java MyProg` weiß, wo es beginnen soll.

Bei funktionalen Programmiersprachen ist naheliegend, dass der Aufruf eines Programms in der Anwendung einer Funktion liegt. Ein bisschen flexibler ist es, wenn man das auf einen Ausdruck verallgemeinert. Ein Ausdruck alleine hängt allerdings in der Luft, wenn man ihm nicht die Definitionen der Namen mitgibt, die in ihm verwendet werden. Damit kommen wir dann ganz natürlich zu folgender Festlegung.

Definition (Programm)

Ein ***Programm*** ist eine Gruppe zusammen mit einem Ausdruck. Wir notieren das in der Form

$$\begin{array}{l}\text{PROGRAM } \mathit{MyProgram} = \{\, \text{ITEM } a_1 = \ldots \\ \qquad\qquad\qquad\qquad\quad \vdots \\ \qquad\qquad\qquad\qquad\quad \text{ITEM } a_n = \ldots \\ \qquad\qquad\qquad\qquad\quad \text{EXEC } e \qquad\quad \}\end{array}$$

Dabei sind natürlich alle Operatoren erlaubt, die wir für Gruppen eingeführt haben bzw. in Kapitel 5 noch einführen werden, insbesondere also USE, EXTEND, IMPORT etc. Für den Ausdruck e gelten dann die üblichen Scoping-Regeln, das heißt, das Programm muss bezüglich der verwendeten Namen abgeschlossen sein.

Anmerkung: Man sieht sofort, dass diese Notation im Prinzip genauso aussieht wie LET $a_1 = \ldots, \ldots, a_n = \ldots$ IN e. *Allerdings würde die Verwendung dieser Notation aus dem Bereich des Programmierens-im-Kleinen auf der Ebene von Packages und Strukturen wohl eher verwirren als helfen. Trotzdem bleibt festzuhalten, dass beide Notationen praktisch das Gleiche beschreiben.*

Der Aufruf eines solchen Programms kann ganz einfach durch Angabe des Namens geschehen:

```
>
> MyProgram
```

Wir können die Flexibilität erhöhen, indem wir Programme ohne EXEC-Ausdruck zulassen. Dann geben wir den Ausdruck e erst auf der Kommandozeile an:

```
>
> MyProgram "foo(3.1, 7.2) + 1"
```

Bedingung ist allerdings, dass der Compiler auch einen Interpretermodus besitzt. Beide Features können auch verbunden werden, indem man Default-Ausdrücke zulässt:

$$\begin{aligned}\text{PROGRAM } \mathit{MyProgram} = \{\, &\text{ITEM } a_1 = \ldots \\ &\vdots \\ &\text{ITEM } a_n = \ldots \\ &\text{DEFAULT EXEC } e\,\}\end{aligned}$$

Dieser Ansatz stellt einen guten Kompromiss zwischen zwei Wünschen dar. Einerseits möchte man nicht auf einen starren Namen für die Startfunktion festgelegt sein, andererseits sollte man nicht bei jedem Aufruf des Programms zeilenlange Auswahlangaben machen müssen.

4.7.2 ... und was ist mit den Programmdateien?

Ist das alles nicht Augenwischerei? Abgehobene akademische Fingerübung? Schließlich reden wir die ganze Zeit über Konzepte für das Programmieren-im-Großen, also über Programme mit einigen Hunderttausend oder gar Millionen Zeilen Code. So hübsch unsere Packages und Librarys auch konzeptuell sein mögen, wenn ihr Verhältnis zu den Dateien und Directorys des Betriebssystems nicht klar ist, nützt das Ganze überhaupt nichts.

Im klassischen OPAL gibt es eine strenge Korrelation zwischen den Strukturnamen und den Dateien, in denen sie gespeichert sind. In JAVA ist es nicht viel anders, die Package-Namen spiegeln sogar direkt die Directory-Struktur wider. Microsofts .NET-Ansatz ist hier schon flexibler: Mit Hilfe so genannter „Metadaten“, die in „Manifesten“ enthalten sind, wird die Verbindung zum Dateisystem relativ flexibel und robust hergestellt. Deshalb verfolgen wir bei unseren entsprechenden Überlegungen diesen Ansatz weiter.

Abbildung 4.2 skizziert die Situation. Die Programmwelt besteht im Wesentlichen aus geschachtelten Gruppen. Im Betriebssystem werden diese in Dateien gespeichert, die wiederum in Ordnerhierarchien eingebettet sind. Die beiden Strukturierungen werden weitgehend entflochten und unabhängig voneinander gemacht, indem ein so genanntes *Manifest* im Stil von .NET zwischengeschaltet wird, also eine Datei, die die entsprechenden Zuordnungen enthält.

In wesentlichen Grundzügen wurde dieses Prinzip in JAVA und .NET eingeführt. Allerdings weichen wir in zwei Punkten davon ab: Eine der etwas merkwürdigeren Designentscheidungen bei JAVA ist es, dass man eigentlich nie genau weiß, was alles in einem Package ist. Denn man kann jederzeit eine weitere Datei mit einer neuen Klasse C schreiben und diese mittels der

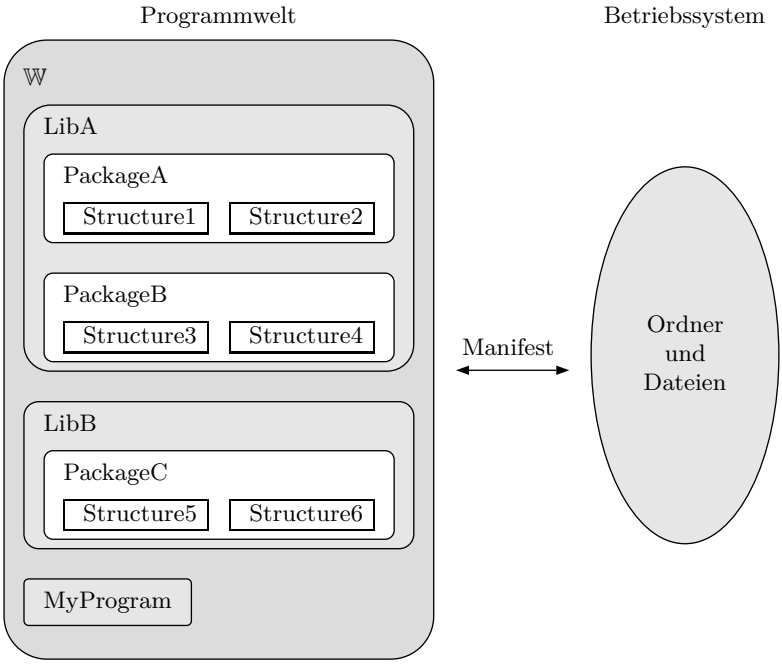


Abb. 4.2: Programmwelt und reale Welt

Anweisung `package p` nachträglich zum Bestandteil des Packages `p` machen. Zur Philosophie der Gruppen gehört, dass sie gerade über die Menge der auf ihnen definierten Selektionen charakterisiert sind. Also sollte diese auch klar umrissen sein. Und bei .NET ist das Manifest auf die generierten Code-Dateien bezogen. Wir wollen es aber zum Bestandteil der Programmtext-Dateien machen.

Das führt auf ein Design wie es in Abbildung 4.3 illustriert ist. Eine Bibliothek L_1 enthalte zwei Packages P_1 und P_2, die ihrerseits vier Strukturen A_1, A_2, S_1 und S_2 enthalten. Wegen ihrer Größe werden diese Gruppen auf vier Programmdateien verteilt.

Jede Datei kann eine oder auch mehrere Gruppen enthalten, aber jede Gruppe muss *vollständig* in einer Datei enthalten sein. Dabei heißt *vollständig*, dass alle Items aufgelistet sein müssen. Mit dem Schlüsselwort EXTERNAL kann ausgedrückt werden, dass die tatsächliche Definition des Items sich in einer anderen Datei befindet. (Es ist allerdings nicht sinnvoll, hier auch zu fixieren, welche Datei das ist; erfahrungsgemäß würde das zu einem sehr starren und unhandlichen System führen.) In unserem Beispiel wird in der Datei A festgelegt, dass die drei Items $L_1.P_1$, $L_1.P_2.S_1$ und $L_1.P_2.S_2$ sich jeweils in anderen Dateien befinden.

Bei der Definition solcher externer Items muss dann mit Hilfe des Schlüsselworts INSIDE gesagt werden, zu welcher Gruppe sie gehören. Der Compiler

File A:

```
LIBRARY L1 = {
  PACKAGE P1 EXTERNAL
  PACKAGE P2 = { STRUCTURE S1 EXTERNAL
                 STRUCTURE S2 EXTERNAL
                 TYPE t1 = ...          }
}
```

File B:

```
INSIDE LIBRARY L1
  PACKAGE P1 = { STRUCTURE A1 EXTERNAL
                 STRUCTURE A2 EXTERNAL }
```

File C:

```
INSIDE PACKAGE L1.P1
  STRUCTURE A1 = { TYPE t1 = ...
                   DEF f1 ...    }
  STRUCTURE A2 = { TYPE t2 = ...
                   ...           }
```

File D:

```
INSIDE PACKAGE L1.P2
  STRUCTURE S1 = { ... }
  STRUCTURE S2 = { ... }
```

Abb. 4.3: Verteilung von Gruppen auf Programmdateien

kann dann prüfen, ob dort entsprechende externe Items vorgesehen sind. Zu diesem Zweck kann der Compiler ein *Manifest* im Stil von .NET erzeugen, in dem die Zuordnungen von Dateien zu Gruppen und Items enthalten ist.

Dieses Design ist einerseits sicher und robust, weil man keine Items in Gruppen hineinschmuggeln kann. Andererseits ist es aber auch sehr flexibel, weil offen bleibt, in welcher Datei die externen Definitionen sich befinden. Auf diese Weise kann man sehr leicht Implementierungen austauschen.

5

Operatoren auf Gruppen (Morphismen)

Es kommt auf dieser Welt viel darauf an,
wie man heißt: der Name tut viel.
Heine (Reisebilder Italien)

Mit dem Konzept der Gruppen haben wir ein Sprachmittel geschaffen, das in uniformer Weise Modularisierung auf allen Ebenen erlaubt. Aus dem Software-Engineering weiß man aber, das die dadurch gegebenen Scopingregeln für praktisches Arbeiten noch nicht ausreichen: Für das Programmieren-im-Großen braucht man noch weitere Möglichkeiten des flexiblen Umgangs mit Gruppen.

Erfreulicherweise wurde das auch aus theoretischer Sicht untermauert: Im Bereich der Algebraischen Spezifikation wurde das Konzept der so genannten Signatur- und Spezifikationsmorphismen entwickelt, das genau den Bedürfnissen des Software-Engineerings gerecht wird.

5.1 Vererbung (EXTEND)

Die Notation USE ... aus Kapitel 4 liefert nur eine *Schreibabkürzung* für die Verwendung von Selektoren. Betrachten wir nochmals Programm 4.6 auf Seite 84. In der Struktur S steht USE B; das erlaubt, innerhalb der Struktur S den partiellen Namen b zu verwenden, weil er als der entsprechende Selektor von B erkannt wird. Die Struktur B enthält ihrerseits die Anweisung USE A; das führt aber *nicht* dazu, dass S jetzt auf indirektem Weg auch die Selektoren von A kennen würde. Mit der Notation USE A wird A *nicht* zum Bestandteil von B.

Manchmal möchte man aber gerade diesen Effekt haben! Der wesentliche Zweck bei der Bildung von *Subklassen* in objektorientierten Sprachen wie JAVA liegt gerade in der Hinzunahme von weiteren Variablen und Methoden. Dieses Feature hat unter dem Schlagwort *Vererbung* nahezu kulthaften Status in der objektorientierten Programmierung erhalten. Da es sich um ein durchaus

nützliches Ausdrucksmittel handelt, das dazu noch sehr einfach realisierbar ist, nehmen wir es auch hier auf. Das artifizielle Programm 5.1 zeigt den Effekt.

Programm 5.1 Vererbung

```
GROUP A = { ITEM a = ...}
GROUP B = { EXTEND A
            ITEM b = ... a ...}
GROUP C = { USE B
            ITEM c = ... a ... b ...}
```

Durch EXTEND A wird tatsächlich die Gruppe A zum Teil von B. Formal können wir das folgendermaßen definieren: Die Gruppe B erzeugt jetzt die Selektionen $a(B)$ und $b(B)$. Dazu kommt die Definition DEF $a(B) = a(A)$. Das heißt, die beiden Selektoren $a(\_)$ in B und C werden beide zu $a(B)$ aufgelöst.

Natürlich ist es nach wie vor möglich, in B oder C auch die volle Selektion $A.a$ – was ja $a(A)$ bedeutet – zu schreiben. Damit wird dann das Item aus A direkt angesprochen. Im Wesentlichen wird hier also das realisiert, was in Sprachen wie JAVA mit dem Schlüsselwort `super` erreicht wird, nämlich der direkte Zugriff auf eine Komponente der Superklasse.

Für das Programmieren-im-Großen, also für Strukturen, Packages etc., ist dieses Vorgehen vertraut und als nützlich bekannt. Aber es funktioniert auch beim Programmieren-im-Kleinen.

TYPE *Point2* = { $x = Real$, $y = Real$ }
TYPE *Point3* = { EXTEND *Point2*, $z = Real$ }

Sogar auf einzelne Punkte können wir das Konzept herunterbrechen.

p: *Point2* = { $x = 3.1$, $y = 4.75$ }
q: *Point3* = { EXTEND p, $z = 1.0$ }

Anmerkung: Es gibt noch einen zweiten Aspekt von Vererbung: die Bildung von Subtypen. Darauf gehen wir in Kapitel 7 näher ein. In vielen objektorientierten Sprachen werden beide Aspekte miteinander verschmolzen, was manchmal zu konzeptuellen Irritationen führt.

5.2 Signatur-Morphismen

Namensräume sind Mengen von benannten Elementen. Damit bieten sich zwei elementare Operationen an:

- Bildung von Teilmengen
- Umbenennung

Beide Aktivitäten sind spezielle Fälle des mathematischen Konzepts der so genannten ***Signatur-Morphismen***. Dieses Konzept ist insbesondere im Bereich der algebraischen Spezifikationssprachen intensiv untersucht worden. Die Sprache SPECWARE [6] stellt sogar eine Reihe von expliziten Operatoren bereit, mit denen sich solche Morphismen konstruieren und anwenden lassen, aber auch in CASL sind ähnliche Konzepte eingebaut [19, 101].

Wir zeigen die Operatoren hier für USE; sie sind aber genauso auf EXTEND anwendbar.

5.2.1 Restriktion (ONLY, WITHOUT)

Die ***Restriktion*** auf eine Teilmenge des Namensraums kann man ganz einfach durch Notationen realisieren, wie sie im Programm 5.2 illustriert sind. Man beachte, dass b immer noch in der langen Form $S.b$ erreichbar ist. (Deshalb kann man Restriktionen bei USE gut benutzen, um Namenskonflikte zu vermeiden.)

Programm 5.2 Restriktions-Morphismus (positiv, ONLY)

```
GROUP S = { ITEM a = ... ITEM b = ... ITEM c = ... }
GROUP T = { USE S ONLY a c                          -- analog für EXTEND
            ITEM x = ... a ... ~~b~~ ... S.b ... c }
```

Wenn man alle Komponenten bis auf wenige Ausnahmen importieren möchte, kann eine Negativliste lesbarer sein. Das wird in Programm 5.3 illustriert. (In HASKELL würde dies in der Form IMPORT S HIDING b geschrieben.)

Programm 5.3 Restriktions-Morphismus (negativ, WITHOUT)

```
GROUP S = { ITEM a = ... ITEM b = ... ITEM c = ... }
GROUP T = { USE S WITHOUT b                         -- analog für EXTEND
            ITEM x = ... a ... ~~b~~ ... S.b ... c }
```

5.2.2 Renaming (RENAMING)

Oh, ich habe meinen guten Namen verloren!
Shakespeare (Othello)

Manchmal kommt man nicht ohne ***Renaming*** aus. Denn wenn man z. B. eine Bibliotheksstruktur entwirft, lässt sich nicht vorhersehen, wo sie überall

benutzt werden wird. Dann kann es geschehen, dass die gewählten Identifier mit Namen an der Verwendungsstelle kollidieren. Dieser Konflikt lässt sich nur an der Verwendungsstelle auflösen. Deshalb braucht man Notationen wie in Programm 5.4 gezeigt.

Programm 5.4 Renaming-Morphismus (RENAMING)

```
GROUP S = { ITEM a = ...  ITEM b = ...  ITEM c = ... }
GROUP T = { USE S RENAMING (a AS u) (b AS v)          -- analog für EXTEND
            ITEM x = ... u ... v ... c ... ~~a~~ ... ~~b~~ ... S.a ... S.b }
```

Semantisch gesehen bedeutet dies, dass mit dem USE-Term die drei Selektionen $u(S)$, $v(S)$ und $c(S)$ bereitgestellt werden, wobei für u und v noch die Definitionen DEF $u(S) = a(S)$ und DEF $v(S) = b(S)$ erfolgen. (Das ist notwendig, weil u und v ja sonst nirgends definiert sind.) Damit werden dann die Muster $u(\_)$, $v(\_)$ und $c(\_)$ erkennbar. Die vollen Namen $a(S)$ und $b(S)$ sind natürlich nach wie vor bekannt.

Häufig ist es lesbarer, die Umbenennungen in einer kompakten Liste zu schreiben. Deshalb erlauben wir die gleichwertige Notation

```
GROUP T = { USE S RENAMING (a, b) AS (u, v)    -- analog für EXTEND
  ...                                        }
```

Anmerkung: Man könnte auch versuchen, auf den RENAMING*-Operator zu verzichten und stattdessen den Parametermechanismus zu strapazieren. Im obigen Beispiel von Programm 5.4 würde das bedeuten, die Struktur S parametrisiert zu schreiben als* GROUP $S(a, b) = \{\ldots\}$*; dann könnte man in T schreiben* USE $S(u, v)$*. Dieses Vorgehen hat aber mehrere Nachteile. Zum einen ist es intuitiv und auch von der Theorie her merkwürdig, etwas als Parameter anzugeben, was im Rumpf definiert wird. Zum anderen ist der Parametermechanismus aber auch pragmatisch ungeeignet; denn man müsste ja alle Komponenten, die man irgendwo vielleicht einmal umbenennen möchte, in die Parameterliste aufnehmen.*

Mathematisch gesehen ist das Vorgehen mit Renaming übrigens verwandt zu den so genannten Fibrations in der Kategorientheorie. (Eine etwas genauere Diskussion findet sich in [118].)

5.2.3 Kombination und Verwendung von Morphismen

Beide Arten von Morphismen – die Restriktion A ONLY $a_1 \ldots a_n$ und die Umbenennung A RENAMING $(a_1$ AS $b_1)$ $\ldots$ $(a_n$ AS $b_n)$ – induzieren jeweils eine Funktion Φ, die auf die Gruppe A angewandt wird und eine neue Gruppe liefert. Im ersten Fall ist $\Phi(A)$ eine Teilmenge von A, im zweiten Fall ist $\Phi(A)$ eine systematische Umbenennung von A.

Wie alle Funktionen lassen sich auch diese Morphismen komponieren. Ein Beispiel sehen wir in Programm 5.5. Das Ergebnis ist hier eine Grup-

pe $\Psi(\Phi(S))$, wobei Φ der Restriktions-Morphismus ist und Ψ der Renaming-Morphismus.

Programm 5.5 Kombinierte Morphismen

```
GROUP S = { ITEM a = ...  ITEM b = ... ITEM c = ... }
GROUP T = { USE S ONLY a c RENAMING (a AS u)          -- analog für EXTEND
            ITEM x = ... u ... c ... ~~a~~ ... ~~b~~ ... S.a ... S.b }
```

Natürlich lassen sich die Morphismen auch in der anderen Reihenfolge kombinieren, wie das Programm 5.6 zeigt. Man beachte, dass sich die Restriktion jetzt auf den bereits geänderten Namen u beziehen muss.

Programm 5.6 Kombinierte Morphismen

```
GROUP S = { ITEM a = ...  ITEM b = ... ITEM c = ... }
GROUP T = { USE S RENAMING (a AS u) ONLY u c          -- analog für EXTEND
            ITEM x = ... u ... c ... ~~a~~ ... ~~b~~ ... S.a ... S.b }
```

Aus Bequemlichkeit und wegen der besseren Lesbarkeit sollte man auch eine verkürzte Notation für kombinierte Morphismen anbieten, wie sie in Programm 5.7 illustriert ist.

Programm 5.7 Kombinierte Morphismen (abgekürzt)

```
GROUP S = { ITEM a = ...  ITEM b = ...  ITEM c = ... }
GROUP T = { USE S ONLY (a AS u) c                     -- analog für EXTEND
            ITEM x = ... u ... c ... ~~a~~ ... ~~b~~ ... S.a ... S.b }
```

5.2.4 Vererbung mit Modifikation

In der objektorientierten Programmierung wird es als ganz wichtig angesehen, dass man bei der Vererbung kleinere Modifikationen vornehmen darf. Als typische Beispiele werden in solchen Diskussionen gerne Szenarien der folgenden Bauart herangezogen: Die Klasse *Tier* hat eine Methode *fliehen*. Bei der Subklasse *Säugetier* wird das als *rennen* implementiert, bei der Subklasse *Vogel* als *fliegen*. Das Ganze geht bei den Subklassen von *Vogel* noch weiter: Bei *Emu* wird die Methode *fliehen* wieder zu *rennen*, und bei *Pinguin* wird sie zu *tauchen*.

Zwar wirkt dieser Vergleich – denn das ist das obige Szenario ja nur – auf den ersten Blick bestechend, aber in der Praxis der Softwareentwicklung zeigen sich doch Probleme. Vor allem geschieht es relativ oft, dass die Modifikation nicht mit Absicht, sondern aus Versehen passiert. Das ist besonders häufig der Fall, wenn Software im Rahmen der Maintenance oder Weiterentwicklung fortgeschrieben wird – was wohl auf über achtzig Prozent der tatsächlichen Programmierarbeit zutrifft. Damit die Vererbung nicht vom Segen zum Fluch mutiert, muss man diese Schwierigkeiten in den Griff bekommen.

Das Hauptproblem scheint darin zu liegen, dass man nicht gewarnt wird, wenn man eine bestehende Methode überschreibt. Dieser Schritt müsste also explizit benannt werden, um größere Sicherheit zu bekommen. Erfreulicherweise – manche würden sagen: notwendigerweise – passt diese Forderung auch besser zu einem konzeptuell sauberen Vorgehen. Denn eine solche saubere Lösung besteht darin, modifizierende Vererbung als eine Kombination von harmloser Vererbung und Restriktion zu sehen. Das ist in Programm 5.8 illustriert.

Programm 5.8 Vererbung mit Modifikation

```
GROUP A = { ITEM a1 = ...  ITEM a2 = ...  ITEM a3 = ... }
GROUP B = { EXTEND A WITHOUT a2
            ITEM a2 = ... }
```

Da diese sichere Lösung ohne großen Aufwand möglich ist, kann man das implizite Überschreiben von Operationen als Fehler („*duplicate definition*“) behandeln. Der – im Vergleich zum Gewinn an Sicherheit winzige – Preis besteht darin, jeweils so etwas wie WITHOUT a_2 explizit hinzuschreiben.

Beispiel 5.1 (Gleichheitstest)

Die Verwendung des standardmäßigen Gleicheitstests „ = “ auf *Real*-Zahlen ist im Allgemeinen aufgrund von Rundungsfehlern sinnlos. Deshalb sollte man die Gleichheit entsprechend umdefinieren:

```
STRUCTURE MyNumbers = {
  EXTEND Float WITHOUT =
  ITEM _ = _ : (Real × Real → Real) = λx, y • |x − y| < 10^−8
}
```

Damit können Algorithmen unverändert in lesbarer Form mit Vergleichen IF $x = y$ THEN ... geschrieben werden.

Dieses Konzept ist auch beim Programmieren-im-Kleinen nützlich. Überraschend oft braucht man leicht modifizierte Tupel oder Gruppen. Das heißt,

es ist ein Wert x gegeben und man braucht einen Wert y, der von x nur in einer oder zwei Komponenten abweicht. Als ein simples Beispiel betrachten wir ein Datum (wobei wir annehmen, dass entsprechende Subtypen *Day*, *Month* und *Year* von *Int* gegeben sind):

$$\text{TYPE } \mathit{Date} = \{\, \mathit{day} = \mathit{Day},\ \mathit{month} = \mathit{Month},\ \mathit{year} = \mathit{Year} \,\}$$

Wir können eine Funktion schreiben, die ein Datum um ein Jahr verschiebt (wobei wir das Problem der Schaltjahre ignorieren).

$$\begin{aligned}&\text{FUN } \mathit{nextYear}\colon \mathit{Date} \to \mathit{Date}\\ &\text{DEF } \mathit{nextYear}(\mathit{date}) = \{\, \mathit{day} \;=\; \mathit{date}.\mathit{day},\\ &\qquad\qquad\qquad\qquad\ \ \mathit{month} \;=\; \mathit{date}.\mathit{month},\\ &\qquad\qquad\qquad\qquad\ \ \mathit{year} \;=\; \mathit{date}.\mathit{year} + 1 \,\}\end{aligned}$$

Wenig elegant ist dabei die längliche Auflistung der unveränderten Komponenten – und zwar nicht nur wegen des Schreibens, sondern vor allem wegen des schlechteren Lesens: die eigentliche Information wird hinter einer Fülle von Banalitäten verborgen. Deshalb lohnt sich für solche Fälle die Einführung einer Spezialnotation:

$$\text{DEF } \mathit{nextYear}(\mathit{date}) = \mathit{date} \text{ BUT } \{\, \mathit{year} = \mathit{date}.\mathit{year} + 1 \,\}$$

Wie wir eben schon im Zusammenhang mit der modifizierenden Vererbung gesehen haben, ist dies nur die Kombination der beiden Operatoren EXTEND und WITHOUT:

$$\begin{aligned}&\text{DEF } \mathit{nextYear}(\mathit{date}) = \{\, \text{EXTEND } \mathit{date} \text{ WITHOUT } \mathit{year}\\ &\qquad\qquad\qquad\qquad\qquad\ \ \mathit{year} = \mathit{date}.\mathit{year} + 1 \qquad\quad \,\}\end{aligned}$$

5.3 Geheimniskrämerei: Import und Export

The vast majority of our imports
come from outside the country.
George W. Bush

Die bisher gezeigten Konzepte bestechen durch ihre Einfachheit. Das reicht auch völlig aus für die Anforderungen des Programmierens-im-Kleinen. Aber aus dem Software-Engineering ist bekannt, dass man für das Programmieren-im-Großen eine filigranere Kontrolle braucht, insbesondere das so genannte *Geheimnisprinzip*.

Erstaunlicherweise wird dieses Geheimnisprinzip in manchen Programmiersprachen – z. B. JAVA – nicht symmetrisch gesehen, sondern einseitig. Doch Abbildung 5.1 macht deutlich, dass die Schnittstelle zwischen einer Gruppe (bzw. einem Fragment) und ihrem (seinem) Kontext in beiden Richtungen gleichartig wirken sollte.

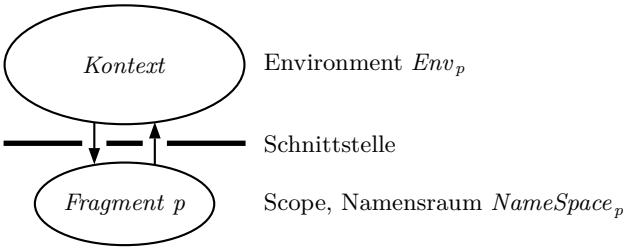


Abb. 5.1: Kontext und Schnittstelle eines Programmfragments

5.3.1 Schutzwall nach außen – Export (PRIVATE, PUBLIC)

Bei unseren Tupeln, Strukturen, Packages etc. haben wir einen Interessenkonflikt. Einerseits sollten sie – es geht ja um große Programme – auf jeden Fall Scoping-Mechanismen zum Schutz lokaler Namen bereitstellen. Andererseits brauchen wir diese lokalen Namen aber als Selektoren.

Die Auflösung dieses Konflikts ist aus Sprachen wie JAVA bekannt: Man spendiert weitere Schlüsselwörter, um die sichtbaren von den geschützten Namen zu unterscheiden. Damit können wir dann z. B. eine Struktur folgender Art schreiben.

STRUCTURE $Map(\alpha, \beta) = \{$
  TYPE $Map = Seq\ (Pair(\alpha, \beta))$
  PRIVATE TYPE $Seq\ \gamma = \ldots$
  PRIVATE TYPE $Pair = \ldots$
...
}

Hier wird im Endeffekt verborgen, dass es sich beim Typ *Map* um eine Sequenz von Paaren handelt. Noch klarer wird das, wenn man das als verteilte Deklaration schreibt:

STRUCTURE $Map(\alpha, \beta) = \{$
  ITEM Map: *Type*
  PRIVATE DEF $Map = Seq\ (Pair(\alpha, \beta))$
  PRIVATE ...
...
}

Damit kann sich von außen niemand auf die „innere“ Struktur des Typs *Map* beziehen und es ist gefahrlos möglich, später auf eine wesentlich effizientere Implementierung umzusteigen. (Natürlich muss es Funktionen geben, die es erlauben, mit den Maps zu arbeiten, ohne die Sequenzen und Paare zu sehen.)

Aus Sprachen wie JAVA ist als Gegenstück zu PRIVATE das Schlüsselwort PUBLIC bekannt. Das wird deshalb gebraucht, weil die Sichtbarkeitsregeln

etwas anders sind als bei uns. Wenn dort z. B. zwei Strukturen A und B in einem Package sind, dann kennen sie gegenseitig ihre Items, ohne dass USE A bzw. USE B gesagt werden muss. Außerhalb des Packages gilt das nicht mehr. Unser systematischerer Ansatz, der gleichartige Scopingregeln über beliebige Hierarchien hinweg etabliert, kommt ohne dieses Schlüsselwort aus.

Es gibt allerdings einen anderen Aspekt, der ein solches Schlüsselwort wieder nützlich macht. Oft hat man große Teile eines Packages oder einer Struktur, die alle mit PRIVATE zu annotieren wären. Dies behindert die Lesbarkeit empfindlich. Deshalb sollte man solche Items zusammenfassen können. Und dann braucht man auch eine Möglichkeit, die Annotation wieder aufheben zu können. Schematisch sieht das dann so aus wie in Programm 5.9.

Programm 5.9 Export: Private und öffentliche Items

```
GROUP A = { PUBLIC PART
              ITEM a1 = ...
              ITEM a2 = ...
            PRIVATE PART
              ITEM a3 = ...
              ITEM a4 = ...
            PUBLIC PART
              ⋮                }
```

Anmerkung: In manchen Sprachen, z. B. OPAL*, wird diese Einteilung fest erzwungen. Der* PUBLIC PART *wird durch das Schlüsselwort* SIGNATURE *gekennzeichnet, der* PRIVATE PART *durch das Schlüsselwort* IMPLEMENTATION*. Beide müssen sogar in getrennten Dateien stehen. Diese letzte Restriktion entspricht aber heute nicht mehr den Prinzipien modernen Software-Engineerings. In* HASKELL *wird der sichtbare Teil dadurch gekennzeichnet, dass er dem Modulnamen in einer so genannten Interfaceliste folgt:* `module (≪Exportliste≫) where ≪Definitionen≫`*. Diese Art von Listen tendiert aber in der Praxis dazu, sehr lange und schwer lesbar zu werden, vor allem wenn im Falle von Overloading noch Typannotationen nötig werden.*

Semantisch bedeutet die Einführung von privaten Items, dass der Namensraum $NameSpace_p$ in zwei Versionen auftritt. Innerhalb des Scopes $Scope_p$ enthält er alle Namen, außerhalb des Scopes enthält er nur noch die *exportierten* Namen, also diejenigen Items die nicht als privat gekennzeichnet sind.

5.3.2 Schutzwall nach innen – Import

Für die Robustheit des Programmierprozesses braucht man eine Kontrolle darüber, was in einer Struktur, einem Package etc. tatsächlich aus der Umgebung benutzt wird. Das wird von der USE-Notation nicht geleistet, weil man innerhalb des Codes noch mit voll qualifizierten Selektionen der Art $A.a$ auf ein Item a zugreifen kann, selbst wenn USE A WITHOUT a gesagt wurde.

Wir verwenden das Schlüsselwort IMPORT um auszudrücken, dass die entsprechende Gruppe gegen alles andere abgeschirmt ist. Die Situation lässt sich schematisch darstellen wie in Programm 5.10.

Programm 5.10 Import

```
GROUP A = { ITEM a = ... ITEM b = ... ITEM c = ... }
GROUP G = { IMPORT A ONLY a c
            ITEM x = ... a ... ~~A.b~~ ... ~~A.b~~ ... c }
```

Semantisch bedeutet dies, dass innerhalb von G alles aus der Umgebung verschattet wird, mit Ausnahme der beiden Selektionen $A.a$ und $A.c$. Die lokalen Namen von G bleiben natürlich verfügbar.

Aus pragmatischen Gründen verbindet man den Import automatisch mit dem Effekt von USE; das heißt, die Items der importierten Struktur können auch gleich in abgekürzter Form benutzt werden.

Formal lässt sich dieses Konzept ganz einfach beschreiben. Wir betrachten wieder den Matching-Prozess zur Namensvervollständigung aus Abschnitt 4.3. Jetzt werden zuerst aus dem Environment alle Namen herausgefiltert, die nicht zum Import gehören (im obigen Beispielprogramm 5.10 also alles, was nicht zu A gehört). Danach wird das Matching wie üblich durchgeführt. Man beachte, dass dabei auch volle Namen der Art $P.S.x$ ausgeschlossen werden, wenn P oder $P.S$ nicht im Import liegen.

Man beachte weiterhin, dass der wesentliche Effekt beim Import also nicht in dem liegt, was geholt wird, sondern in dem was nicht mehr geholt werden darf (nämlich alles andere).

Ein pragmatisches Problem mit der Typisierung

Mit unserem rigorosen Ansatz beim Import haben wir ein kleines, aber lästiges Problem geschaffen. Betrachten wir folgendes Beispiel:

```
STRUCTURE Foo = {
  IMPORT Sequence ONLY Seq length
  ...
  DEF f(s: Seq) = ... length(s) ...
  ...
}
```

Die Funktion *length* hat als Ergebnistyp *Nat*. Dieser Typ ist hier aber nirgends mit importiert worden. Hier muss der Compiler so gestaltet werden, dass es zu keinen unnötigen Fehlermeldungen kommt. Auf keinen Fall darf aber dem Programmierer zugemutet werden, hier zusätzliche Importe vorzunehmen, nur damit die Typisierung aller benutzten Items ausdrückbar wird.

5.4 Generizität: Funktionen, die Gruppen liefern

Dass man Funktionen schreiben kann, bei denen als Ergebnis Gruppen entstehen, ist nichts Neues. Bei Tupeln kennt man das gut, bei Records sieht es fast genauso aus:

FUN *shift*: $Real \times Real \to Point \to Point$
DEF $shift(dx, dy)(p) = \{\, x = p.x + dx,\ y = p.y + dy \,\}$

Diese Anwendung erfolgt in Programmiersprachen derart standardmäßig, dass man nicht einmal einen besonderen Namen für diese Art von Funktionen eingeführt hat.

Anders ist das bei „höheren“ Arten von Gruppen. Bei Strukturen, Packages oder gar Bibliotheken kommt man nicht so ohne weiteres auf die Idee, sie mittels Funktionen zu generieren. Dabei gibt es ganz einfache und durchaus bekannte Anwendungen.

```
STRUCTURE Sequence α = {
  TYPE Seq = ...
  FUN length: Seq → Nat = ...
  FUN ft: Seq → α = ...
      ⋮
}
```

Offensichtlich ist *Sequence* hier eine Funktion, die als Argument einen Typ nimmt und als Ergebnis eine Struktur abliefert. Konsequenterweise ist es dann möglich, diese Funktion in vielfältiger Weise zu applizieren.

```
STRUCTURE NatSeq = Seq Nat
STRUCTURE Word = { EXTEND Seq Char  ... }
```

Die Parameter müssen nicht unbedingt Typen sein; auch Parametrisierung mit Werten ist möglich, wie das Beispiel von (m, n)-Matrizen zeigt.

```
STRUCTURE Matrix (m, n) α = {...}
```

Mögen diese Beispiele gerade noch akzeptabel erscheinen, so bedarf es bei den noch höheren Arten von Gruppen schon eines gewaltigen Maßes an Phantasie. Dabei gibt es nützliche Beispiele; sie sind nur – in Ermangelung geeigneter Sprachkonstrukte – ganz anders gelöst worden. Compiler-Generatoren könnte man z. B. sehr elegant und kompakt folgendermaßen schreiben:

```
PACKAGE CompilerTools(Grammar) = {
  STRUCTURE Lexic =   { FUN scanner = ...
                        VAL symboltable = ... }
  STRUCTURE Syntax =  { FUN parser = ...
                        TYPE AbsTree = ... }
      ⋮
}
```

So etwas überfordert alle gängigen Programmiersprachen. Die pragmatische Lösung besteht darin, die Grammatik in einer Datei abzulegen, aus der die verschiedenen Generator-Tools sie jeweils holen. Das ist nichts anderes, als Parameterübergabe „von Hand“ auf dem Umweg über das Betriebssystem.

Anmerkung 1: Auch bei diesen generischen Records, Strukturen, Packages etc. sieht man deutlich, dass ein wesentlicher Aspekt in unserer Diskussion noch fehlt: die Typisierung. Wie schon mehrfach angekündigt, wird uns das in den Kapiteln 6 bis 9 noch intensiv beschäftigen.

Anmerkung 2: Die naheliegende Erkenntnis, dass sich der Funktionsgedanke auf die Modularisierungsebene übertragen lässt, hat interessanterweise kaum den Weg in die Programmiersprachen gefunden. Wenn das Phänomen dort anzutreffen ist, dann unter Namen wie Templates oder generische Module, für die dann jeweils sehr spezifische semantische Beschreibungen gegeben werden. Dass es sich nur um eine weitere Anwendung der Funktionsidee handelt, wird kaum einmal gesagt.

Eine bemerkenswerte Ausnahme ist die Sprache ML*. Hier gibt es dieses Konzept unter dem Begriff Functor:*

FUNCTOR *«name»* (*«parameter»*: *«signature*$_{param}$*»*): *«signature*$_{result}$*»* =
STRUCT *«definitions»* END

Der Parameter ist eine ganze Struktur, die über ihre Signatur (s. Kapitel 9) typisiert wird. Das Ergebnis ist ebenfalls eine Struktur, die über eine entsprechende Signatur typisiert wird. In OPAL *ist es ähnlich; allerdings werden die Parameter nicht als Struktur, sondern als Liste einzelner Items angegeben. Aber sowohl* ML *als auch* OPAL *schränken das Konzept stark ein: Diese Art von Funktionen existieren nur auf der obersten Ebene und es gibt keine weiteren Möglichkeiten als sie zu definieren und zu instanziieren. Das heißt, Strukturen und ihre Signaturen sind keine* First-class citizens.

Nochmals: Schlüsselwörter

Bei den generischen Typen, Strukturen, Packages etc. gibt es ein kleines Problem mit der Wahl unserer spezifischen Schlüsselwörter. Betrachten wir das obige Beispiel der generischen Sequenzen.

Eigentlich müsste es FUN *Sequence* $\alpha = \{\dots\}$ heißen, weil es sich um eine Funktion handelt. Aber aus pragmatischen und historischen Gründen ist es vernünftig, hier im Schlüsselwort das Resultat der Applikation hervorzuheben und deshalb von STRUCTURE zu sprechen. (Hier bewährt sich, dass unsere Schlüsselwörter nur Abkürzungen und damit sehr flexibel zu handhaben sind.)

6

Typen

Das Wesen der Dinge ist schwerlich so flach wie die Mehrzahl der Köpfe, die ihm auf den Grund gekommen zu sein glauben.
Otto Liebmann

Die zentrale Frage zum Typkonzept von Programmiersprachen lautet natürlich: *„Welche Typkonstrukte bietet die Sprache?“* Um das einschätzen zu können, muss man aber eine zweite Frage mit stellen. *„In welchen allgemeinen Rahmen ist das Typkonzept der Sprache eingebettet?“* Was sich hinter dieser zweiten Frage verbirgt, soll im Folgenden als erstes skizziert werden. Danach werden die gängigen Typkonzepte von (funktionalen) Programmiersprachen im Einzelnen behandelt. Weitergehende Fragen wie Subtypen, Polymorphie, abhängige Typen oder Typklassen werden erst in den darauf folgenden Kapiteln diskutiert werden.

6.1 Generelle Aspekte von Typen

Zuerst wenden wir uns der zweiten der obigen Fragen zu, also dem Rahmen, in den die Typisierungskonzepte der Sprache eingebettet sind.

6.1.1 Die Pragmatik: Statische oder dynamische Typprüfung?

Als Erstes muss man klären, *wann* die Typkorrektheit überprüft wird. Grundsätzlich gibt es zwei Möglichkeiten (vgl. Abbildung 6.1):

- *Dynamische Prüfung* (*run-time check*). Bei der Ausführung des Programms wird bei jedem Aufruf $f(e)$ einer Funktion $f\colon A \to B$ überprüft, ob der Argumentwert e dem geforderten Typ A genügt und ob das Resultat dem Typ B genügt.
  Vorteil: Man kann ein sehr reiches und ausdrucksstarkes Typsystem einführen, das alle methodisch erwünschten Konzepte enthält.

Nachteil: Das Verfahren ist sehr ineffizient, weil zur Laufzeit sehr viele zusätzliche Tests ausgeführt werden. Außerdem können die Typprüfungen ihrerseits auf Fehler führen, z. B. Division durch Null, Zahlüberlauf, unendliche Berechnungen etc.

- *Statische Prüfung* (*compile-time check*). Der Compiler analysiert das Programm um sicherzustellen, dass jeder Funktionsaufruf die Typisierung erfüllt. Man kann das als den einfachsten Grenzfall einer Programmverifikation auffassen.
  Vorteil: Die Prüfung findet nur einmal bei der Übersetzung statt und belastet die Ausführung des Programms nicht.
  Nachteil: Da letztlich ein Beweis stattfindet, müssen die Typisierungskonzepte so einfach und eingeschränkt sein, dass dieser Beweis tatsächlich vollautomatisch und effizient durchgeführt werden kann. Außerdem dürfen die Typprüfungen keinen Absturz des Compilers bewirken, selbst dann nicht, wenn der Benutzer ein falsches System von Typen programmiert hat.

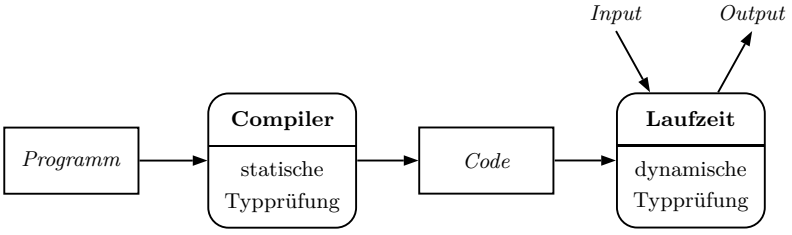


Abb. 6.1: Statische und dynamische Typprüfung

In der Praxis wird aus Effizienzgründen in fast allen Sprachen die statische Typprüfung gewählt – mit den entsprechenden Beschränkungen für die Ausdrucksmächtigkeit der Sprache. Die dynamische Prüfung ist seltener zu finden, am ehesten noch bei Arrays, wenn die Einhaltung der Indexgrenzen sichergestellt wird, und bei gewissen Aspekten von objektorientierten Sprachen. Zu den wenigen Sprachen mit dynamischer Typprüfung gehören z. B. ERLANG, SMALLTALK, PYTHON und – mit Einschränkungen – LISP und JAVA.

Wir interessieren uns vor allem für moderne Programmierparadigmen, also für möglichst ausdrucksstarke und nützliche Konzepte. Davon sollten wir uns nicht durch Probleme der effizienten Implementierbarkeit ablenken lassen. Also gehen wir grundsätzlich von *dynamischer Typprüfung* aus. Dabei betrachten wir es als *Optimierungsproblem*, möglichst viele dieser Prüfungen in den Compiler zu verlagern. (Dies wird in der Literatur auch als *Type erasure* bezeichnet.) Außerdem gibt es immer noch die Möglichkeit – wie beim `assert`-Statement von JAVA–, die Prüfung bei Bedarf auszuschalten.

Mit anderen Worten: *Wir gehen davon aus, dass in einer modernen Sprache beides stattfindet: statische und dynamische Typprüfung.*

6.1.2 *Reflection*: Typen als *„First-class citizens“*

Bei nahezu allen gängigen Programmiersprachen beschränkt sich die Rolle der Typen darauf, gewisse Standardfehler in Programmen abzufangen, indem Variablen, Konstanten, Prozeduren etc. „typisiert“ werden. Mit objektorientierten Sprachen wie JAVA oder dem .NET-Framework von Microsoft ist aber eine Erweiterung dieser Rolle erfolgt: Typen – dort *Klassen* genannt – können im Programm selbst zur Laufzeit in Berechnungen einbezogen werden. Dazu dienen Klassen wie *Object* oder *Class*, und das spiegelt sich im generellen Konzept der so genannten *Reflections* wider [131, 132, 41]. Eine weitere Motivation ergibt sich bei der Verwendung von *Garbage Collection*: hier wird zur Laufzeit eine Menge an Typinformation benötigt.

Dabei geschieht eigentlich etwas ganz Einfaches. Wenn man einen Compiler für eine Sprache wie z. B. OPAL schreibt, dann gibt es in diesem Compiler Datenstrukturen, mit deren Hilfe die im jeweiligen Programm vorkommenden Typen repräsentiert werden. Diese Datenstrukturen werden in traditionellen Compilern nach der Übersetzung weggeworfen. Das Reflection-Prinzip basiert darauf, dass man sie nicht wegwirft, sondern dem Programm erlaubt, zur Laufzeit auf sie zuzugreifen. Durch JAVA ist diese Idee inzwischen in der objektorientierten Welt verbreitet, in [42] wurde sie experimentell für die funktionale Sprache OPAL realisiert.

Aber wir wollen diese Idee nicht nur als compilertechnisches Phänomen hinnehmen, sondern das Prinzip auf konzeptueller Ebene konsequent zu Ende denken. Das heißt, wir betrachten Typen als *„First-class citizens“*. Das führt dazu, dass zwei Sichten auf Typen in unseren Programmen gleichberechtigt nebeneinander stehen (vgl. Abbildung 6.2):

- ***Intensionale Sicht***. Jeder Typ ist ein eigenständiges „Ding“ und somit im Programm benutzbar und verarbeitbar.
  Wenn also im Compiler oder bei der Anwendung von Reflection der Typ selbst als Datenstruktur codiert ist, dann spiegelt das seine intensionale Sicht wider.
- ***Extensionale Sicht***. Jeder Typ charakterisiert eine gewisse Menge von Werten.
  Wenn wir also sagen, dass mit einer Schreibweise wie $5 : Nat$ der Wert 5 als ein Element der natürlichen Zahlen charakterisiert wird, dann haben wir eine extensionale Sicht von *Nat* eingenommen.

Anmerkung: Dieses Begriffspaar wird in der Logik und Philosophie z. B. von Carnap und Wittgenstein betrachtet. Wir passen die Begriffe hier technisch auf unsere programmiersprachlichen Konzepte an.

Betrachten wir z. B. den elementaren Typ Int. In intensionaler Sicht ist damit genau ein Ding bezeichnet (das nur zwei Eigenschaften hat, nämlich Int zu heißen und ein Typ zu sein). In extensionaler Sicht steht Int für die Menge der ganzen Zahlen (bzw. für die Menge der 32-Bit-Zahlen). Das ist ein bedeutender Unterschied zur klassischen Mathematik. Wenn dort z. B.

von der Menge $Prim = \{x \in \mathbb{N} \mid prim\ x\}$ die Rede ist, dann hat der Name *Prim* keinerlei Relevanz und keinerlei eigenständige Bedeutung; er bezieht seine Existenzberechtigung ausschließlich aus der Menge, für die er steht. In einem Ausdruck wie $29 \in Prim$ ist *Prim* deshalb auch nichts anderes als eine Kurzschreibweise für die entsprechende Menge.

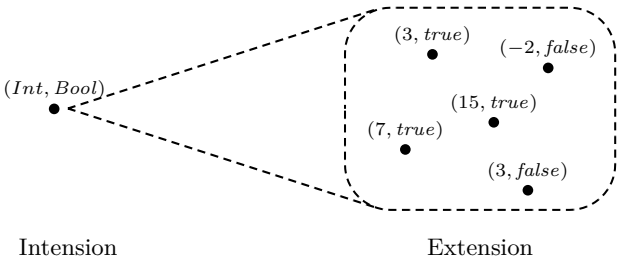


Abb. 6.2: Intensionale und extensionale Sicht

In Abbildung 6.2 wird dieser Zusammenhang illustriert. Ein Tupeltyp wie z. B. $(Int, Bool)$ ist in *intensionaler* Sicht einfach nur ein Paar bestehend aus den beiden Symbolen Int und $Bool$. Genau dieses Paar wird vom Compiler im Rahmen der Typprüfung bearbeitet, und dieses Paar wird bei Verwendung von Reflection-Techniken auch in die Laufzeit übernommen. In *extensionaler* Sicht steht dieser Tupeltyp dagegen für eine Menge von Werten, und zwar für alle Paare, deren erste Komponente (extensional) ein Element von Int ist und deren zweite Komponente (extensional) ein Wert von $Bool$ ist.

Der Übergang zwischen beiden Sichten erfolgt, wenn wir Typisierungen angeben wie z. B. $(-2, true) : (Int, Bool)$. Links und rechts vom Doppelpunkt stehen jeweils intensionale Elemente, einmal ein Tupel bestehend aus den Symbolen $-$ 2 und *true* und einmal ein Tupel bestehend aus den Symbolen Int und $Bool$. Aber die *Semantik* des Ausdrucks basiert auf der extensionalen Sicht: Die Elemente links sind in den Mengen rechts enthalten.

In einem modernen Ansatz, der auch Konzepte wie Reflection einbezieht, muss man sich beim Arbeiten mit Typen also immer darüber im Klaren sein, dass sie zwei Seiten haben. Wenn wir so etwas schreiben wie

TYPE $Prim = (x : Nat \mid prim\ x)$

(die Notation wird in Kapitel 7 im Zusammenhang mit Subtypen genauer erklärt werden) dann hat der Typ *Prim* eine Doppelnatur. Zum einen kann er in einem Ausdruck der Art $29 : Prim$ verwendet werden, was dem $29 \in Prim$ der Mathematik entspricht. Zum anderen existiert *Prim* aber auch als ein eigenständiges Ding, das zwei Eigenschaften repräsentiert: „*Nat* sein“ und „*prim* sein“. Die Repräsentation dieser beiden Fakten ist außerdem in geeigneter Weise programmiersprachlich codiert – üblicherweise als Paar bestehend aus

dem Basistyp *Nat* und der Funktion *prim* –, so dass man sie abfragen und auch manipulieren kann.

6.1.3 Intensionalität, Reflection und der Compiler

Compilertechnisch haben die obigen Beobachtungen eine einfache Konsequenz: *Jeder Wert ist mit seinem Typ attributiert, d. h., er ist ein Paar bestehend aus dem Wert selbst und dem Typ.*

Typen als Attribute sind ein Standardkonzept des Compilerbaus, das während der Kontextanalyse eine zentrale Rolle spielt. Wenn diese Attribute – wie z. B. in JAVA oder .NET – auch zur Laufzeit erhalten bleiben, werden einige Fragen, die sonst nur Compilerbauer beschäftigen, auch für Programmierer relevant:

- *Wie stellt man Typen dar?* JAVA macht es sich hier sehr einfach: alles wird über Strings dargestellt; der Rest ist das Problem des Programmierers. In einem systematischen Ansatz muss man hier bessere Lösungen anbieten – und die sind aus Compilern ja bekannt.
- *Wie kann man möglichst hohe Effizienz erreichen?* Dabei bedeutet Effizienz zweierlei: Zum einen soll der Laufzeitcode möglichst wenig mit Typprüfungen belastet werden. Zum anderen weiß man, dass die Produktivität der Programmierer wesentlich steigt, wenn viele Fehler bereits im Compiler abgefangen werden. Beides läuft darauf hinaus, dass man möglichst gute Optimierungstechniken für die Typprüfung braucht.
- *Kann ein Wert mehr als einen Typ haben?* Das ist eine subtile Frage, die in der objektorientierten Programmierung unter dem Stichwort *mehrfache Vererbung* heftig diskutiert wird.

Wir werden im Folgenden immer wieder über diese Beziehung zwischen Werten und ihren Typen reden müssen. Deshalb führen wir eine spezielle Notation dafür ein.

Definition (Wert mit annotiertem Typ)

Um auszudrücken, dass ein Wert a zur Laufzeit den Typ T hat, schreiben wir $a^{\langle T \rangle}$. (Im Zusammenhang mit Summentypen wird es auch Listen von annotierten Typen geben; s. Abschnitt 6.5.)

Beispiele für diese Notation sind etwa $7^{\langle Int \rangle}$, $3.2^{\langle Real \rangle}$, $\texttt{"}Hallo\texttt{"}^{\langle String \rangle}$, $(1.2, true)^{\langle Real, Bool \rangle}$, $(1, 7, 4)^{\langle Array(0..2)Real \rangle}$ usw.

Anmerkung: Wenn man alle Werte zur Laufzeit tatsächlich in dieser Form mit ihren Typen annotiert, geht sehr viel Effizienz verloren. Deshalb ist es eine wesentliche Optimierungsaufgabe, diese Annotationen wegzulassen, wo immer sie nicht gebraucht werden. Technisch lässt sich das so machen, dass die Paare $(Wert, Typ)$ *compilerintern jeweils in zwei getrennte Variablen aufgeteilt werden. Dann identifiziert eine standardmäßige Datenflussanalyse diejenigen Variablen, die nie gebraucht werden.*

6.1.4 Typen sind auch nur Terme

Im Gegensatz zur klassischen Typtheorie, wo Typen üblicherweise semantisch und kalkülorientiert betrachtet werden, nehmen wir eine mehr pragmatische, compilerorientierte Sicht ein. Wir sehen Wertausdrücke ebenso wie Typausdrücke als *Terme* an (die compilerintern als Bäume realisiert sind); diese Terme sind manchmal durch das Symbol „ : “ miteinander verbunden, wodurch jeweils noch größere Terme entstehen. Zum Beispiel ist

$$(sin(x), cos(x)) : (Real \times Real)$$

ein Term mit zwei Subtermen, nämlich $(sin(x), cos(x))$ und $(Real \times Real)$, die durch den Operator „ : “ miteinander verbunden sind.

Terme dieser Bauart müssen – wie alle anderen Programmterme auch – geeignet interpretiert werden; das kann zur Compilezeit geschehen oder erst zur Laufzeit. Diese Interpretation muss mit der semantischen Sicht von Typen verträglich sein, das heißt, sie muss den Regeln eines entsprechenden Typkalküls genügen. Mit anderen Worten, wir nehmen hier eine intensionale Sichtweise ein, die mit einer entsprechenden extensionalen semantischen Sicht kompatibel sein muss.

Diese Betrachtungsweise führt uns direkt auf das Konzept der so genannten *Coercion Semantics*, bei der Typisierung letztlich als Typanpassung verstanden wird; darauf gehen wir im Detail in Abschnitt 7.1.3 ein.

6.1.5 Typdeklarationen: Typsynonyme oder neue Typen?

Eine ebenso interessante wie subtile Frage ist, ob eine Typdeklaration nur einen synonymen Namen einführt (equi-recursive view [117]) oder ob ein ganz neuer Typ entsteht (iso-recursive view [117]). Die erste Sichtweise erscheint auf den ersten Blick intuitiver, sie erweist sich aber in der Praxis als komplizierter. Wir beantworten diese Frage mit einem entschlossenen sowohl-als-auch: Letztlich hängt es davon ab, was bei einer Typdeklaration auf der rechten Seite steht. In einer Konstruktion der Bauart

$$\text{TYPE } A = «rhs»$$

ist der Typname A in der Tat *synonym* zu seiner rechten Seite «*rhs*». Die Frage, ob ein neuer Typ entsteht, hängt damit von der rechten Seite ab. In Fällen wie

$$\text{TYPE } Dollar = Int \quad -- \textit{ Typsynonym (äquivalente Namen)}$$

ist *Dollar* wirklich nur ein anderer Name für Int. Wenn wir dagegen schreiben

$$\text{TYPE } Parser = (String \to Tree) -- \textit{ Typsynonym (Namen und Ausdruck)}$$

ist *Parser* eine Kurzschreibweise für den Ausdruck $(String \to Tree)$. Noch deutlicher wird der Effekt, wenn wir Konstruktoren hinzunehmen, also einen so genannten „Sum-of-products“-Typ definieren (s. Abschnitte 6.4 und 6.5):

TYPE *Dollar* = *dollar*(*value*: *Int*) −− *neuer Typ*
TYPE *Parser* = *parser*(*String* → *Tree*) −− *neuer Typ*

Hier haben wir auf der rechten Seite jeweils einen neuen Typ geschaffen, dessen Werte mit Hilfe des angegebenen Konstruktors generiert werden müssen. Zu diesen neuen (anonymen) Typen sind die Namen *Dollar* bzw. *Parser* dann synonym.

Anmerkung: In den Sprachen ML *und* HASKELL *ist das anders gelöst. Dort benutzt man unterschiedliche Schlüsselwörter. Sowohl in* ML *als auch in* HASKELL *verwendet man für Typsynonyme das Schlüsselwort type. Für neue Typen verwendet man in* ML *das Schlüsselwort datatype und in* HASKELL *die Schlüsselwörter data oder newtype.* OPAL *kennt keine Typsynonyme.*

type dollar = *int* (∗ *Synonym in* ML ∗)
type Dollar = *Int* −− *Synonym in* HASKELL

datatype dollar = *Dollar of int* (∗ *neuer Typ in* ML ∗)
data Dollar = *Dlr Int* −− *neuer Typ in* HASKELL

ML *besitzt darüber hinaus noch ein weiteres Schlüsselwort, mit dem man so genannte abstrakte Typen einführen kann:*

abstype nat = *Nat of int* (∗ *abstrakter Typ in* ML ∗)
with «definitions»

Hier werden im Wesentlichen die Konstruktoren verschattet, d. h., die interne Repräsentation der Werte verborgen.

6.1.6 *Kinding*: Typen höherer Stufe

Wenn wir Typen nur als Terme behandeln, dann stellt sich die Frage nach der Typisierung dieser Typterme. In der Typtheorie spricht man hier von *Kinding*. Wir werden dieses Thema im Detail in Kapitel 9 behandeln; aber zwei Aspekte müssen wir schon vorab ansprechen:

- Um Operatoren auf Typen selbst typisieren zu können, brauchen wir die Klasse *Type* aller Typen. Damit ist dann eine Spezialnotation wie

  TYPE *Point* = ...

  nur eine Abkürzung und Lesehilfe für die elementarere Konstruktion

  ITEM *Point*: *Type* = ...

- Wenn wir Operatoren auf Typen ihrerseits typisieren wollen, dann ist das Schlüsselwort TYPE offensichtlich nicht angebracht. Also verwenden wir ein anderes Schlüsselwort:

  KIND *Seq*: *Type* → *Type*

  Diese Zeile besagt, dass *Seq* eine Funktion ist, die Typen in Typen abbildet. (Wir werden diese Sichtweise von polymorphen Funktionen in Kapitel 8 einführen.)

Definition (KIND, *Type*)

Das generelle Schlüsselwort für die Annotation der Art eines Items ist KIND.

KIND «*Item*» : «*Art*»

Das Schlüsselwort TYPE ist nur der Spezialfall für die Annotation von Werten mit ihren Typen.

Die Klasse aller Typen bezeichnen wir mit *Type*.

6.1.7 Mehrfachtypisierung (Mehrfachvererbung)

Kann ein Wert mehrere Typen haben? Diese Frage wird bei traditionellen Programmiersprachen normalerweise nicht gestellt; dort hat ein Wert immer genau einen Typ. Aber sobald man Subtypen zulässt (s. Kapitel 7) stellt sich die Frage neu. Und weil die so genannte *Vererbung* bei objektorientierten Sprachen nur ein neues Wort für Subtypen ist, bekommt die Frage sogar eine hohe praktische Relevanz: Man kennt sie unter dem Namen *Mehrfachvererbung*. Spätestens bei der Betrachtung von so genannten *Typklassen* in Kapitel 9 (die den *Interfaces* von JAVA entsprechen) wird die Mehrfachtypisierung sogar zu einer zwingenden Notwendigkeit.

Aus diesen Gründen müssen wir *Mehrfachtypisierung* generell zulassen. Das bedeutet, dass z. B. ein Wert x die drei Typen haben kann

PROP x: *Nat*
PROP x: *Odd*
PROP x: *Prim*

Was bei diesem Beispiel noch ziemlich artifiziell wirkt, kann bei Tupeltypen durchaus praktischen Sinn haben (s. Abschnitt 7.3) und wird bei Typklassen zu einem essenziellen Konzept (s. Kapitel 9). Deshalb sehen wir auch eine Notation für Mehrfachtypisierung vor:

```
TYPE x:  Nat & Odd & Prim                    -- Mehrfachtypisierung
KIND Int:  Numeric & Eq & Ord & Printable    -- Mehrfachklassen
```

6.2 Elementare Typen

Jedes praktische Typisierungskonzept muss mit elementaren ***Grundtypen*** beginnen, aus denen dann alle weiteren Typen aufgebaut werden. Diese elementaren Grundtypen werden als gegeben vorausgesetzt. Im Bereich der Programmiersprachen hat sich ein Standardkanon herauskristallisiert, den wir in Tabelle 6.1 zusammengefasst haben.

Dabei gibt es allerdings Variationen und Probleme. Die wichtigste Frage ist, ob die zahlartigen Typen jeweils die mathematischen Mengen $\mathbb{N}$, $\mathbb{Z}$ und $\mathbb{R}$ meinen oder nur die entsprechenden beschränkten Maschinenzahlen. Für

Bool	die Wahrheitswerte *true*, *false*
Nat	die natürlichen Zahlen
Int	die ganzen Zahlen
Real	die reellen Zahlen
Char	die „Zeichen“, meist das Alphabet der ASCII- oder Unicode-Zeichen
String	Texte, also Folgen von Zeichen

Tab. 6.1: Elementare Grundtypen

den letzteren Fall sehen einige Sprachen noch Varianten mit unterschiedlichen Bitlängen vor (*Byte*, *Short*, *Long*, *Float*, *Double*). Wir werden uns hier den Luxus leisten, diese Frage offen zu lassen.

Anmerkung: Im Bereich der so genannten Algebraischen Spezifikation *wurden Techniken entwickelt, mit denen man auch solche Basistypen vollständig formal definieren kann, ohne auf die Pragmatik der Maschinen zurückgreifen zu müssen. (In den Büchern [43, 44] und [19, 101] kann man Genaueres dazu finden.)*

6.3 Aufzählungstypen

Neben den vordefinierten Grundtypen braucht man manchmal auch selbst definierte Basistypen; dazu dienen die ***Aufzählungstypen***. Bei der Beschreibung gewisser Filme wäre z. B. der folgende Typ nützlich:

TYPE *Attitude* = { *good*, *bad*, *ugly* }

Diese Deklaration führt einen neuen Typ *Attitude* ein, der die drei Werte *good*, *bad* und *ugly* umfasst. Diese Werte sind grundsätzlich *linear geordnet*,[1] und zwar in der Reihenfolge ihrer Aufschreibung, also $good < bad < ugly$.

Definition (Aufzählungstyp)

Ein ***Aufzählungstyp*** wird in folgender Form geschrieben:

TYPE $T = \{ a_1, \ldots, a_n \}$

Dabei werden sowohl der Typ T als auch seine Elemente a_i eingeführt. Die Elemente sind in der Reihenfolge ihrer Aufschreibung geordnet und es gibt

[1] Das ist eine pragmatische Design-Entscheidung. Sie resultiert aus Erfahrungen mit der Sprache OPAL. Dort hat sich das Konzept mit ungeordneten Werten als unpraktikabel erwiesen, sobald die Zahl der Werte größer wurde. HASKELL löst das Problem anders: Hier wird die Aufzählung zwar wie in OPAL als Summentyp dargestellt, kann aber als Instanz der Typklasse `Enum` (s. Kapitel 9) gekennzeichnet werden.

neben den üblichen Ordnungsrelationen noch die Operationen *min*, *max*, *succ* und *pred*.

Mit der obigen Typdeklaration *Attitude* werden also automatisch einige Konstanten und Funktionen eingeführt:

```
VAL min: Attitude                          -- min = good
VAL max: Attitude                          -- max = ugly
FUN succ: Attitude → Attitude              -- Nachfolger (partiell)
FUN pred: Attitude → Attitude              -- Vorgänger (partiell)
FUN _ < _: Attitude × Attitude → Bool      -- Ordnung
    ⋮
```

Die drei Pünktchen stehen für die übrigen Ordnungsrelationen wie $=$, $\leq$ etc.

Interessanterweise haben weder OPAL noch ML oder HASKELL wirkliche Aufzählungstypen. Alle drei Sprachen emulieren dieses Konzept über das Mittel der Summentypen (s. Abschnitt 6.5 weiter unten). Allerdings hat die Erfahrung gezeigt, dass bei größeren Aufzählungstypen – die z. B. in der lexikalischen Analyse des Compilerbaus vorkommen – die lineare Ordnung unverzichtbar ist. Das wird von den Summentypen üblicherweise nicht geleistet. Auch JAVA hat in der Version 1.5 die Idee der Aufzählungstypen eingeführt, allerdings – aufgrund der bestehenden Klassen- und Vererbungsregeln – auf sehr komplexe und in den Details schwer durchschaubare Weise.

Im weiteren Verlauf des Buches werden wir drei spezielle (jeweils einelementige) Aufzählungstypen sehr häufig verwenden:

```
TYPE Empty  = { ◊ }     -- leere Sequenz, leerer Baum, ...
TYPE Fail   = { fail }  -- Failure (bei Maybe oder Exception)
TYPE Void   = { void }  -- der Wert "Nichts" (bei Ein-/Ausgabe)
```

Der Typ *Empty* spielt bei funktionalen Datenstrukturen eine ähnliche Rolle wie der Nullpointer bei imperativen Datenstrukturen; er hat allerdings den großen Vorteil, typisiert zu sein.

Den Typ *Fail* braucht man z. B., um bei Berechnungen das Auftreten von Fehlersituationen zu signalisieren. (Für die reellen Zahlen sieht der IEEE-Standard dafür den Wert *NotANumber*, kurz *NaN*, vor. Bei anderen Datentypen muss man ihn bei Bedarf individuell hinzufügen.) Im Zusammenhang mit *Exceptions* werden wir diesen Typ um spezifischere Angaben erweitern.

Insbesondere bei Ein-/Ausgabe hat man immer wieder Situationen, in denen Operationen gar kein Ergebnis abliefern, sondern nur einen „Effekt“ haben. Das signalisiert man mit dem Pseudotyp *Void*. (Dies ist analog zu JAVA; in HASKELL wird dies durch das leere Klammerpaar () bezeichnet.)

6.4 Tupel- und Gruppentyp

Eine der natürlichsten Arten, neue Typen einzuführen, ist die Bildung von geordneten Tupeln $(a_1, \ldots, a_n)$. In der Mengenlehre spricht man hier vom *kartesischen Produkt*. In Kapitel 4 haben wir zwar schon diskutiert, dass Tupel nur ein Spezialfall des allgemeineren Konzepts der Gruppen sind; da sie in der Programmierung aber häufiger benutzt werden, beginnen wir mit den Tupeltypen.

6.4.1 Tupeltyp

Eines der elementarsten Beispiele für Tupel ist die Repräsentation von Punkten im $\mathbb{R}^2$. Für die Notation von Tupeltypen findet man zwei äquivalente Notationen, die eine mehr intensional im Stil von Programmiersprachen, die andere eher extensional in der Tradition der Mathematik:

TYPE $Point = (Dist, Angle)$ −− *elementare Tupelnotation (intensional)*
TYPE $Point = Dist \times Angle$ −− *alternative Infix-Notation (extensional)*

Mit diesen Definitionen gelten z. B. für den Punkt $p\colon Point = (1.4142, 45)$ folgende Typaussagen:

$p\colon Point$ −− *gemäß Definition*
$(1.4142, 45)\colon Point$ −− *Einsetzen*
$(1.4142, 45)\colon (Dist, Angle)$ −− *Einsetzen*
$1.4142\colon Dist \wedge 45\colon Angle$ −− *Distribution*

Dabei haben wir folgende Prinzipien benutzt: 1. Dinge, die gleich sind, können füreinander eingesetzt werden; das gilt für Typen ebenso wie für Werte. 2. Typisierung distribuiert über Tupel.

Definition (Tupel, Selektor, Konstruktor)

Ein ***Tupeltyp*** wird in einer der beiden gleichwertigen Formen geschrieben:

TYPE $T = (T_1, \ldots, T_n)$ −− *intensionale Sicht*
TYPE $T = T_1 \times \ldots \times T_n$ −− *extensionale Sicht*

Zusätzlich gibt es noch die Variante mit ***Selektoren***

TYPE $T = (s_1 = T_1, \ldots, s_n = T_n)$

und die Variante mit ***Konstruktor*** und Selektoren:

TYPE $T = cons(s_1 = T_1, \ldots, s_n = T_n)$

Die ***Werte*** von Tupeltypen werden entsprechend geschrieben (für die ersten drei Formen gleich):

VAL $x\colon T = (x_1, \ldots, x_n)$

Bei der Konstruktorvariante schreibt man

VAL $x\colon T = cons(x_1, \ldots, x_n)$

Die Typisierung distribuiert über die Tupelbildung:

$(x_1, \ldots, x_n)\colon (T_1, \ldots, T_n) \quad \Longleftrightarrow \quad x_1\colon T_1 \wedge \ldots \wedge x_n\colon T_n$

Diese Typdeklarationen sind im Grunde genommen Abkürzungen, die eine Reihe von Funktionen induzieren:

```
TYPE T
FUN _: T1 × ... × Tn → T      -- unsichtbarer Konstruktor
FUN cons: T1 × ... × Tn → T   -- expliziter Konstruktor
FUN s1: T → T1                -- Selektor
    ...                       -- ...
FUN sn: T → Tn                -- Selektor
```

Dabei stellt die Version mit dem unsichtbaren Konstruktor den Compiler natürlich vor größere Probleme als die mit einem expliziten Konstruktor; deshalb wird letztere in vielen Sprachen vorgeschrieben.

Mit Hilfe der Klasse *Type* aller Typen können wir die Bildung von Produkttypen selbst auch als entsprechende Operatoren einführen:

```
KIND ( _, ..., _ ): Type × ... × Type → Type
KIND ( _ × ... × _ ): Type × ... × Type → Type
```

Genau genommen handelt es sich allerdings um eine ganze Familie von Operatoren, weil wir eine beliebige Stellenzahl zulassen. Aber man erkennt auch sofort ein weiteres Problem: Zur Definition des Operators „ × “ auf Typen verwenden wir den Operator „ × “ auf der Ebene der Klassen. Wir haben es also mit einem gestuften System zu tun (mehr dazu in Kapitel 9).

Notation in ML, HASKELL und OPAL

In ML könnte unser obiges Beispiel *Point* so aussehen:[2]

```
(* ML - Notation *)
datatype point = Point of dist * angle
```

In ML hat man also einen Konstruktor, aber keine Selektoren. Elemente von Tupeltypen können daher dort nur mit Hilfe musterbasierter Funktionen (s. Abschnitt 1.1.7) bearbeitet werden.

In HASKELL wird nahezu alles in Curry-Notation geschrieben. Das gilt sogar für Tupeltypen (ohne Selektoren).

```
-- HASKELL-Notation
data Point = Point Dist Angle                    -- ohne Selektoren
data Point = Point {dist :: Dist, angle :: Angle}  -- mit Selektoren
```

[2] Wir schließen uns hier der gängigen Konvention an, Konstruktoren in ML mit Großbuchstaben beginnen zu lassen; Typen schreibt man in ML meist klein.

OPAL verwendet die in der obigen Definition angegebene Variante mit Konstruktor und Selektoren:

```
-- OPAL-Notation
DATA point == point(dst: dist, ang: angle)
```

Sind Tupel assoziativ?

Im Zusammenhang mit Tupeln gibt es noch eine weitere Designentscheidung: Gilt die Assoziativität des Tupel-Operators?

$$((A \times B) \times C) \stackrel{?}{=} (A \times B \times C) \stackrel{?}{=} (A \times (B \times C))$$

Falls ja, dann kann man z. B. in der folgenden Situation

```
FUN f: A × B × C → ...
FUN g: ... → A × B
FUN h: ... → C
```

die Funktionsapplikation schreiben:

$$\ldots f(\, g(\ldots),\, h(\ldots)\,) \ldots$$

Das sieht auf den ersten Blick recht elegant aus und passt auch zur Tradition der Mathematik. Deshalb wurde diese Entscheidung z. B. in OPAL so getroffen. Inzwischen hat aber die Erfahrung gezeigt, dass die Assoziativität der Tupelbildung massive Einschränkungen bei Polymorphie und generischen Strukturen nach sich zieht, und dieser Verlust wiegt weit schwerer als der Gewinn bei Funktionsapplikationen der obigen Art.

Festlegung (Tupelbildung ist *nicht* assoziativ)

Für Tupeltypen gilt

$$((A \times B) \times C) \neq (A \times B \times C) \neq (A \times (B \times C))$$

Damit man Applikationen der obigen Art trotzdem schreiben kann – was selten genug gebraucht wird – kann man einen speziellen „Konkatenations"-Operator $\otimes$ einführen, der Tupel flach macht, so dass man z. B. die Gleichheit hat

$$(A \times B) \otimes (C \times D) \quad = \quad (A \times B \times C \times D)$$

Wir werden in Abschnitt 8.2 ein Beispiel sehen, in dem dieser Konkatenations-Operator auf Tupeln tatsächlich notwendig ist.

6.4.2 Gruppentyp

In Kapitel 4 wurde das allgemeinere Konzept der Gruppen eingeführt, für die folgende Notation verwendet wird:

TYPE $Point = \{\, dist = Dist,\ angle = Angle \,\}$
VAL $p\colon Point = \{\, dist = 1.4142,\ angle = 45 \,\}$

Man beachte, dass die Notation konsequenterweise auf der Typebene die gleiche sein muss wie auf der Wertebene. Im einen Fall stehen die Selektoren für Typen, im anderen Fall für Werte. Damit beides zusammenpasst, müssen die Selektoren gleich heißen. (Die Reihenfolge ist aber irrelevant.)

In Analogie zum Vorgehen bei den Tupeltypen distribuiert auch hier die Typisierung über die Gruppenbildung:

$$\begin{array}{ll} & p\colon Point \\ \iff & \{\, dist = 1.4142,\ angle = 45 \,\}\colon \{\, dist = Dist,\ angle = Angle \,\} \\ \iff & 1.4142\colon Dist \quad \wedge \quad 45\colon Angle \end{array}$$

Diese Distributivität lässt sich auch über die Selektion ausdrücken. (Zur Erinnerung: eine Selektionsfunktion wie $dist(p)$ schreiben wir meistens mit Hilfe des „.“-Operators in der Form $p\,.\,dist$.)

$$\begin{array}{ll} & p\colon Point \\ \iff & (p\,.\,dist)\colon (Point\,.\,dist) \quad \wedge \quad (p\,.\,angle)\colon (Point\,.\,angle) \\ \iff & 1.4142\colon Dist \quad \wedge \quad 45\colon Angle \end{array}$$

Diese kurze Diskussion macht deutlich, dass man einen Tupeltyp wie *Point* konsequenterweise mit Gleichheitszeichen schreiben muss.

Definition (Gruppentyp)

Ein ***Gruppentyp*** wird in folgender Form geschrieben:

TYPE $T = \{\, s_1 = T_1,\ \ldots,\ s_n = T_n \,\}$

Die *Werte* des Typs T werden in analoger Form geschrieben:

VAL $x\colon T = \{\, s_1 = a_1,\ \ldots,\ s_n = a_n \,\}$

Es gilt die Distributivität der Typisierung

$$x\colon T \quad \iff \quad x\,.\,s_i\colon T\,.\,s_i \quad (\iff a_i\colon T_i) \qquad \text{für alle } i$$

Diese Gruppentypen gibt es in der Sprache ML, aber nicht in OPAL oder HASKELL. Unser Beispiel *Point* würde in ML folgendermaßen aussehen:

$type\ point = \{\, Dist\colon dist,\ Angle\colon angle \,\} \quad (*\ \text{ML} - Notation\ *)$

Anmerkung: Traditionell wird der Gruppentyp so wie in diesem ML*-Beispiel geschrieben: Der Selektorname und sein Typ werden mit einem Doppelpunkt verbunden. Unsere homogene Sichtweise von Gruppen zusammen mit dem Prinzip der Distributivität der Typisierung über die Gruppenbildung zeigt aber, dass aus konzeptuellen Gründen ein „ = “ richtig ist.*

6.5 Summentypen

Neben der Produktbildung (Tupel und Gruppen) ist die ***Summenbildung*** die zweite Standardkonstruktion, um neue Typen aus gegebenen Typen zu erhalten.[3] Dabei wird ein Typ eingeführt, der mehrere *Varianten* hat. Ein klassisches Standardbeispiel sind geometrische Figuren:

TYPE *Shape* = *Point* | *Line* | *Circle* | *Triangle* | *Rectangle*

Hier sind *Point*, *Line*, ..., *Rectangle* Typen, die anderswo schon definiert wurden. Die Grundidee ist ganz simpel: Wo immer man einen Wert vom Typ *Shape* erwartet, können Werte der Typen *Point*, *Line* etc. angegeben werden: Seien z. B. gegeben

VAL p: *Point* = ...
VAL c: *Circle* = ...

FUN f: *Shape* → ...

Dann dürfen wir Aufrufe schreiben wie

$\ldots f(p) \ldots f(c) \ldots$

Definition (Summentyp)

Ein ***Summentyp***

TYPE $T = T_1 \mid \ldots \mid T_n$

führt einen Typ T ein, dessen Werte genau die Werte der Typen $T_1, \ldots, T_n$ sind; das heißt:

$x\colon T \quad \Longleftrightarrow \quad x\colon T_1 \vee \ldots \vee x\colon T_n$

Unter Verwendung der Klasse *Type* aller Typen kann man diese Konstruktion auch wieder als expliziten Operator einführen:

KIND _ | _: *Type* × *Type* → *Type* −− *kommutativ und assoziativ*

Notation in ML, HASKELL und OPAL

In den Sprachen ML, OPAL und HASKELL werden Summentypen jeweils nur mit expliziten Konstruktorfunktionen eingeführt. So würde man z. B. in ML schreiben

[3] Summentypen haben eine enge Beziehung zu Subtypen, die wir in Kapitel 7 ausgiebig behandeln werden. Da viele Sprachen aber keine Subtypen enthalten, diskutieren wir Summentypen hier als unabhängiges Konzept, auch wenn einige Aspekte dadurch dupliziert werden.

```
datatype shape = Line of point * point                (*ML - Notation*)
               | Circle of point * real
               | Triangle of point * point * point
               | Rectangle of point * real * real
```

In OPAL sieht dieser Typ folgendermaßen aus. Man beachte, dass es in OPAL kein Trennsymbol zwischen den einzelnen Varianten gibt.

```
DATA shape == line(p1: point, p2: point)              -- OPAL-Notation
              circle(center: point, radius: real)
              triangle(p1: point, p2: point, p3: point)
              rectangle(ref: point, wd: real, ht: real)
```

Auch in HASKELL werden Konstruktorfunktionen benötigt, wie üblich in Curry-Notation:

```
data Shape = Line      Point Point          -- HASKELL-Notation
           | Circle    Point Float
           | Triangle  Point Point Point
           | Rectangle Point Float Float
```

Der Zwang, Konstruktorfunktionen zu verwenden, macht dem Compiler das Leben leichter. Denn die jeweils zutreffende Variante ist immer sofort zu erkennen, sei es über explizite Testoperationen wie in OPAL oder über musterbasierte Funktionen wie in ML und HASKELL (und auch in OPAL).

Aber dieser Zwang führt in der Praxis zu einer Fülle von so genannten *Wrapper*-Funktionen, also Konstruktoren für einelementige Tupel, die nur eingeführt werden, weil die Summentyp-Syntax sie fordert. (Ein optimierender Compiler wie der von OPAL eliminiert diese Wrapper zwar auf Code-Ebene wieder, aber die Lesbarkeit stören sie trotzdem.[4])

6.5.1 Was für ein Typ bist du? Dieser Typ!

Wir brauchen eine Notation für den Test der *Typmitgliedschaft*. Das sieht man sofort am Beispiel der folgenden Funktion *area*, die die Fläche einer geometrischen Figur berechnen soll.

```
FUN area: Shape → Real
DEF area(shape) =
    IF shape IS Point  THEN 0
    IF shape IS Line   THEN 0
    IF shape IS Circle THEN ((shape AS Circle).radius)^2 * π
    ...
```

Dieses Beispiel illustriert die Verwendung der zwei komplementären Operationen ***Typtest*** und ***Typanpassung*** (engl.: ***Casting***).

[4] Diese Wrapper sind ähnlich lästig wie die vielen `instanceof-` und Casting-Anweisungen, die z. B. in JAVA (vor der Version 5) bei der Arbeit mit Datenstrukturen notwendig sind.

Definition (Typtest, Typanpassung (Casting))

Sei ein Summentyp gegeben:

TYPE $T = T_1 \mid \ldots \mid T_n$

Dann können wir für Werte x: T die Zugehörigkeit zur Variante T_i testen:

x IS T_i *-- gehört x zur Variante T_i?*

Ebenso können wir Werte x: T in eine Variante *casten*:

x AS T_i *-- mache x zum Wert der Variante T_i (Downcast)*

Diese Operation wird auch als *Downward-Casting* (kurz: *Downcast*) bezeichnet. Sie ist nur definiert, wenn x tatsächlich zur Variante T_i gehört, also der Test x IS T_i *true* liefert. Andernfalls ist das Ergebnis undefiniert.
Aus Symmetriegründen führen wir für x: T_i auch das *Upward-Casting* (kurz: *Upcast*) ein:

x AS T *-- betrachtet x als Element von T (Upcast)*

Wir diskutieren diese Test- und Anpassungsoperatoren hier nicht weiter, weil wir sie in Kapitel 7 im allgemeineren Kontext der Subtypen genauer erörtern werden. Insbesondere die pragmatische Unterscheidung zwischen dem harmlosen (und deshalb vom Compiler automatisch eingeführten) Upcast und dem kritischen Downcast wird dort genauer untersucht.

Vor allem eine Frage wird dabei zu behandeln sein: Wie weit lässt sich das Casting vom Compiler automatisch ergänzen, und wo muss der Programmierer es explizit codieren?

Die Existenz der Testoperationen hat auch Auswirkungen auf die Laufzeitrepräsentation der typannotierten Werte, die in Abschnitt 6.1.3 eingeführt wurde. Beim Upcast muss der Originaltyp erhalten bleiben, weil sonst der Testoperator nicht implementierbar wäre. Sei beispielsweise der Wert p: *Point* gegeben. Dann gilt

$$p^{\langle Point\rangle} \text{AS } Shape \quad \rightsquigarrow \quad p^{\langle Point\rangle\langle Shape\rangle}$$

Hier ist p zwar zu einem Wert des Typs *Shape* geworden, aber Tests wie p IS *Point* oder p IS *Line* können nach wie vor effektiv durchgeführt werden. Beim Downcast wird einfach der zusätzliche Summentyp wieder gestrichen.

Definition (Annotation bei Summentypen, Primärtyp)

Beim Upcast für Summentypen wird bei der Annotation von Werten der Originaltyp beibehalten und der Summentyp *zusätzlich* annotiert. Der Originaltyp wird als ***Primärtyp*** bezeichnet.

6.5.2 Disjunkt oder nicht?

Der klassische Streitpunkt im Zusammenhang mit Summentypen ist die Frage, ob die Varianten *disjunkt* sind oder nicht. Wir haben hier – unserer generellen Intention folgend – das allgemeinere Konzept gewählt, indem wir echte Summenbildung verwenden. Das liegt auch näher an den Arbeiten zur Typtheorie [117]. Das Problem lässt sich am besten anhand von pathologischen Grenzfällen illustrieren.

TYPE $Strange_1 = Dist \mid Angle$
TYPE $Strange_2 = Int \mid Int$

Betrachten wir den ersten Fall. Nehmen wir an, wir haben einen Wert $phi\colon Angle = 180$ und casten ihn nach $Strange_1$; das heißt, wir betrachten den Wert $x\colon Strange_1 = phi$ AS $Strange_1$. Danach führen wir den Test x IS $Dist$ aus. Wenn wir Summentypen als echte Vereinigung auffassen würden, liefe das auf den Test x IS $Real \wedge x \geq 0$ hinaus, der *true* liefert. Damit hätten wir den Winkel 180 Grad in die Distanz 180 verwandelt! *Können wir mit dieser Skurrilität leben? Sicher nicht.*

Das führt uns ganz zwangsläufig auf die obige Festlegung: Wenn wir einen Wert in einen Summentyp casten, dann wird er *mit beiden Typen annotiert* (vgl. Abbildung 6.3).

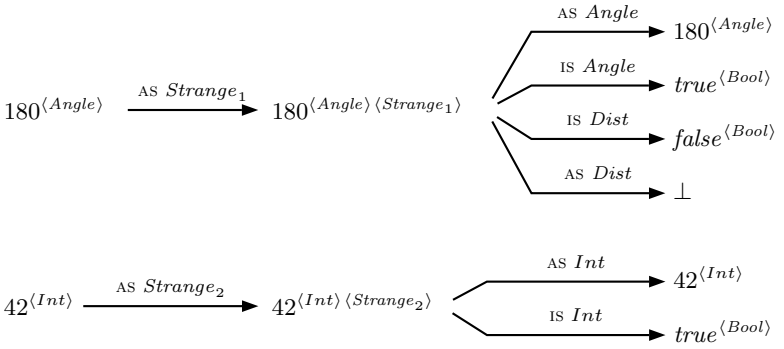


Abb. 6.3: Casting und Variantentest bei Summentypen (zur Laufzeit)

Der ursprüngliche Typ muss erhalten bleiben, weil sonst der Typtest _ IS _ nicht durchführbar wäre. Und der Summentyp wird z. B. gebraucht, wenn der Wert in einen weiteren Summentyp eingebettet würde. Beim Downcast wird – im definierten Fall – einfach die Annotation des Summentyps wieder entfernt. Der undefinierte Fall führt auf einen Fehler, der wie üblich mit dem Pseudoelement $\bot$ (*Bottom*) dargestellt wird. Man beachte, dass mit diesen Definitionen alle Casting- und Testoperationen die richtigen Funktionalitäten haben.

Beim zweiten unserer pathologischen Beispiele zeigt die Rechnung in Abbildung 6.3, dass hier faktisch *Idempotenz* besteht. Denn für den gecasteten Wert gibt es nur die beiden Operationen _ IS Int und _ AS Int.

Festlegung (Disjunktheit bei Summentypen)

Sei ein Summentyp gegeben

TYPE $T = T_1 \mid \ldots \mid T_n$

Dann gibt es für jedes Element x: T genau eine Variante, zu der x gehört:

x: $T \quad \Longleftrightarrow \quad x$ IS T_i für genau ein T_i

Die einzige Ausnahme sind identische Varianten; hier gilt Idempotenz:

$(T \mid T) = T$

6.5.3 Summen mit Typausdrücken

Insbesondere im Zusammenhang mit rekursiven Typen (s. Abschnitt 6.7) hat man sehr oft die Situation, dass eine der Varianten nur einen Wert umfasst. Das lässt sich zwar immer mit Hilfe neu eingeführter Hilfstypen umgehen, aber einen derartigen Umweg zwingend zu fordern, ist keine elegante Lösung. Generell sollten wir also zulassen, dass Summen nicht nur aus benannten Typen gebildet werden, sondern auch aus Typausdrücken. Der Compiler generiert dann eben intern einen geeigneten *anonymen Typ*.

Betrachten wir z. B. einen Typ, der auf *Real* aufbaut und – entsprechend dem IEEE Standard für Gleitpunktzahlen – noch drei Spezialwerte hinzufügt:

TYPE *Float* = *Real*
| $\{\infty, nAn, \propto\}$

Diese drei Werte stehen für *Negativ-Unendlich*, *Not-a-number* und *Positiv-Unendlich*. Damit kann man dann z. B. die Berechnung der Quadratwurzel totalisieren:

FUN *sqrt*: *Real* → *Float*
DEF *sqrt*(x) = IF $x < 0$ THEN *nAn* ELSE ... FI

Durch die obige Typdeklaration wird implizit ein *anonymer Typ* eingeführt, so dass die Deklaration folgender Standardform entspricht

TYPE *Anonym* = $\{\infty, nAn, \propto\}$
TYPE *Float* = *Real*
| *Anonym*

Die gleichen Prinzipien gelten auch für Summen, deren Varianten Tupel- oder Gruppentypen sind. So kann man z. B. Punkte sowohl durch (x,y)-Koordinaten als auch durch Polarkoordinaten darstellen:

$$\begin{aligned}\text{TYPE } Point &= Real \times Real \\ &\mid Dist \times Angle\end{aligned}$$

Hier werden implizit zwei anonyme Typen eingeführt, über denen dann die standardmäßige Summe gebildet wird.

6.5.4 *Syntactic sugar*: Overloading von „ : “

In der Gilde der funktionalen Programmierer gibt es einen Hang zu möglichst knappen Notationen. Daher kann man auf die – sehr experimentelle – Idee kommen, die beiden Operatoren _ IS _ und _ AS _ beide durch _ : _ zu ersetzen, was zu einer hochgradigen Überlagerung des Doppelpunkts führt. Unser obiges Beispiel erhielte dann folgendes Aussehen:

$$\begin{aligned}&\text{FUN } area\colon Shape \to Real \\ &\text{DEF } area(shape) = \\ &\quad \text{IF } shape\colon Point \text{ THEN } 0 \\ &\quad \text{IF } shape\colon Line \text{ THEN } 0 \\ &\quad \text{IF } shape\colon Circle \text{ THEN } ((shape\colon Circle).radius)^2 * \pi \\ &\quad \ldots\end{aligned}$$

Hier sieht man, dass der Doppelpunkt gleich in drei Bedeutungen verwendet wird:

- In einer Deklaration assoziiert der „ : “ den Namen mit seinem Typ.
- In einer Bedingung bezeichnet der „ : “ einen Typtest.
- In den sonstigen Applikationen bezeichnet der „ : “ eine Typanpassung.

Es ist zur Zeit nicht klar, ob Compiler in der Lage sein werden, diese Art von extremer Überlagerung zu verarbeiten. Noch wichtiger ist aber die Frage, ob die Leser eines solchen Programms mit dieser Vieldeutigkeit zurecht kommen. (Wir werden uns in diesem Buch die Freiheit nehmen, experimentell mit dieser extremen Überlagerung zu arbeiten.)

6.6 Funktionstypen

Jetzt fehlt uns nur noch die letzte der fundamentalen Typkonstruktionen, nämlich die Bildung von ***Funktionstypen***. Sie werden gebraucht, wenn wir Funktionalitäten angeben wie z. B.

$$\begin{aligned}&\text{FUN } sin\colon Real \to Real \\ &\text{FUN } shift\colon Real \times Real \to Point \to Point\end{aligned}$$

Funktionstypen werden also – der Tradition der Mathematik folgend – mit dem Infixsymbol _ → _ geschrieben. (In Abschnitt 8.2 werden wir sehen, dass dies nur ein Spezialfall einer allgemeineren Konstruktion ist.)

Definition (Funktionstyp)

Ein ***Funktionstyp*** wird in folgender Form geschrieben:

TYPE $T = (A \to B)$

Dabei sind A und B beliebige Typen, also insbesondere Produkttypen, aber auch wieder Funktionstypen. Die Werte dieses Typs sind alle Funktionen mit Argumenten vom Typ A und Resultaten vom Typ B.

Mit Hilfe der Klasse *Type* lässt sich auch diese Konstruktion als expliziter Operator einführen:

KIND $\_ \to \_ : \mathit{Type} \times \mathit{Type} \to \mathit{Type}$

In der üblichen extensionalen Sicht beschreibt der Typ T die Menge aller Funktionen von A nach B. Wenn man mit Reflection arbeitet, dann braucht man die intensionale Sicht, bei der T zwar auch eine Abbildung ist, aber nur noch eine einelementige Abbildung, die nur für das eine Element A definiert ist und dann liefert $T(A) = B$.

6.7 Rekursive Typen

Wie bei Funktionen ist auch bei Typen die Möglichkeit der *Rekursion* ein entscheidendes Mittel, um hinreichend große Ausdrucksmächtigkeit zu erlangen. Im Gegensatz zu den traditionellen imperativen Sprachen, bei denen Rekursion nur „unter der Hand“ auf dem Umweg über Pointer möglich ist, schreibt man in funktionalen Sprachen die Rekursion direkt hin.

TYPE $\mathit{NatList} = (\mathit{ft} = \mathit{Nat}, \mathit{rt} = \mathit{NatList}) \mid \mathit{Empty}$
TYPE $\mathit{Empty} = \{ \Diamond \}$

Diese rekursiven Typen werden allerdings nur selten in so konkreten Formen wie „Liste natürlicher Zahlen“ hingeschrieben, sondern meistens generisch für beliebige Elementtypen (s. Kapitel 8).

TYPE $\mathit{List}\ \alpha = (\mathit{ft} = \alpha, \mathit{rt} = \mathit{List}\ \alpha) \mid \mathit{Empty}$

Der Typ *Empty* kann auch für den Terminierungsfall anderer Datenstrukturen herangezogen werden.

TYPE $\mathit{Tree}\ \alpha = (\mathit{left} = \mathit{Tree}\ \alpha, \mathit{node} = \alpha, \mathit{right} = \mathit{Tree}\ \alpha) \mid \mathit{Empty}$

Das heißt, der Typ *Empty* übernimmt eine ähnliche Rolle wie der Nullpointer in imperativen Sprachen – allerdings in einer sauberen Typisierung.

Auch indirekte Rekursion ist möglich, wie das folgende Beispiel zeigt.

TYPE *Tree* α = (*node* = α, *subtrees* = *Forest* α)
TYPE *Forest* α = *List* (*Tree* α)

Diese Definition entspricht ziemlich genau der üblichen umgangssprachlichen Beschreibung: Ein Baum ist ein Knoten zusammen mit einem Wald von Unterbäumen. Und ein Wald ist eine Liste von Bäumen.

Notation in ML, HASKELL und OPAL

Die Schreibweisen für rekursive Typen ergeben sich unmittelbar aus den Regeln für Produkt- und Summentypen. Wir illustrieren das am Beispiel der allgemeinen Bäume.

In ML werden zwei verschränkt rekursive Typen mit Hilfe des Schlüsselworts *withtype* miteinander verwoben.

```
datatype 'a tree = Tree of 'a * 'a forest   (* ML - Notation *)
withtype 'a forest = 'a tree list
```

In HASKELL lassen sich rekursive Typen direkt auf Summen- und Produkttypen aufbauen, wobei die Konstruktoren üblicherweise mit Currying notiert werden.

```
data Tree a = Tree a (Forest a)   -- HASKELL-Notation
data Forest a = Forest [Tree a]
```

In OPAL gibt es keine polymorphen Typen; diese müssen auf dem Umweg über generische Strukturen eingeführt werden. Deshalb nehmen wir an, dass die beiden folgenden Typdeklarationen in einer Struktur *Tree*[*data*] stehen:

```
DATA tree == tree(node: data, subtrees: forest)   -- OPAL-Notation
DATA forest == forest(trees: seq[tree])
```

6.8 Wie geht's weiter?

Die Typisierung von Programmiersprachen ist ein großes Gebiet mit vielen Facetten, Variationen und Spielarten. Es gibt die unterschiedlichsten Aspekte von Programmen, die man jeweils als Typsystem darstellen kann. Aus diesen Einzelaspekten lässt sich die Typisierung einer Sprache fast schon baukastenartig zusammenstellen. Und jede Komposition birgt neue tiefe und subtile mathematische Probleme. Der Artikel [30] und die Bücher [117, 118] zeigen das in eindrucksvoller Weise.

Da wir das Typisierungsproblem nicht als theoretische Herausforderung begreifen, sondern es aus der Pragmatik des konkreten Sprachdesigns heraus behandeln wollen, werden wir uns in der Diskussion der folgenden Kapitel auf drei zentrale Erweiterungen der Basiskonzepte konzentrieren:

- Subtypen (Kapitel 7);
- polymorphe und abhängige Typen (Kapitel 8);
- Typklassen (Kapitel 9).

7

Subtypen (Vererbung)

Der Spezialist ist in seinem winzigen Weltwinkel vortrefflich zu Hause; aber er hat keine Ahnung von dem Rest.
Ortega y Gasset

Die Teilmengen-Relation ist ein fundamentales Konzept der mathematischen Mengenlehre. Bei der Programmierung gibt es ein entsprechendes Konzept: *Subtypen*. Allerdings haben die objektorientierten Sprachen hier – wie bei vielen Dingen – ein anderes Wort eingeführt, nämlich *Vererbung*.

Die grundlegende Idee ist intuitiv eingängig. Die natürlichen Zahlen sind ein Subtyp der ganzen Zahlen und die Primzahlen sind ein Subtyp der natürlichen Zahlen. Und so weiter. Leider ergeben sich bei der Umsetzung dieser Idee in konkreten Programmiersprachen eine ganze Reihe von Schwierigkeiten, auf die wir im Folgenden eingehen werden.

Generell gilt: Typsysteme werden wesentlich komplizierter, sobald Subtypen eingebaut werden. Aus der Typtheorie ist bekannt, dass die Komplexität der Inferenzalgorithmen steigt und unter Umständen sogar die Entscheidbarkeit ganz verloren geht. Für genauere Informationen zu diesen theoretischen Fragen verweisen wir wieder auf die entsprechende Literatur, z. B. [30, 117].

7.1 Ein genereller Rahmen für Subtypen

Bevor wir uns die einzelnen Konstruktionen ansehen, mit denen Subtypen eingeführt und verwendet werden können, wollen wir einige generelle Aspekte betrachten.

Grundsätzlich gilt: Welche Arten von Subtyp-Bildung man in einer Sprache zulässt, ist eine *Designentscheidung*. Wie diese Entscheidung getroffen wird, hat massiven Einfluss auf die Ausdrucksmächtigkeit der Sprache und auf die Komplexität des Compilers. Auch wenn – wie für alle Designfragen – die Beurteilung nicht nach richtig/falsch erfolgen kann, sondern eher nach

Kriterien wie gelungen/verkorkst oder elegant/überfrachtet, so gibt es doch Leitlinien, an denen man sich beim Entwurf orientieren kann. Von besonderer Bedeutung ist hier das *Kontextkriterium.*

Definition (Kontextkriterium)

Das ***Kontextkriterium*** besagt: *Der Subtyp kann überall stehen, wo der Supertyp erwartet wird.* Genauer: Sei S ein Subtyp von T. Dann können in jedem *Kontext* $c[\ldots]$, in dem Elemente des Typs T erwartet werden, auch Elemente des Subtyps S stehen.

Wie wir noch sehen werden, kann uns dieses Kriterium immer dann helfen, wenn die „richtige“ Definition von Subtyp nicht sofort evident ist.

Eine zweite Leitlinie ist die *Verträglichkeit* mit den anderen Typisierungsaspekten. Die Subtyp-Relation sollte sich möglichst „natürlich“ in die Welt der Typkonstruktionen einfügen. Das wirkt sich insbesondere bei der Beziehung zu Summentypen aus.

Als Konsequenz dieser Betrachtungsweise erhalten wir eine Antwort auf eine alte Streitfrage: Sind Subtypen (im mathematischen Sinn) *echte Teilmengen* oder nur *Teilmengen modulo einer homomorphen Einbettung*? Im Zusammenspiel mit unseren anderen Typkonstruktionen lautet die Antwort: Subtypen sind Teilmengen modulo homomorpher Einbettung.

Anmerkung: Um nicht missverstanden zu werden: Diese Aussage gilt im Rahmen unserer generellen Behandlung der Typisierungsidee. Bei Sprachen, die eine andere Typphilosophie verfolgen, kann die Antwort dementsprechend anders ausfallen.

7.1.1 Die Subtyp-Relation

Die Subtyp-Relation orientiert sich von der Idee her an der Teilmengenrelation (gegebenenfalls „modulo Isomorphie“). Deshalb verwenden wir das entsprechende mathematische Symbol:

$$Nat \subseteq Int, \qquad Dist \subseteq Real, \qquad Angle \subseteq Real$$

Definition (Subtyp-Relation)

Die ***Subtyp-Relation*** wird folgendermaßen geschrieben

$$S \subseteq T$$

Der Typ S heißt ***Subtyp*** von T und umgekehrt heißt T ***Supertyp*** von S. Die Relation bedeutet, dass für alle Element x gilt:

$$x\colon S \quad \Longrightarrow \quad x\colon T$$

Die Subtyp-Relation ist eine partielle Ordnung, das heißt, sie ist reflexiv, antisymmetrisch und transitiv.

7.1.2 Typanpassung (*Casting*)

Als zentrale Leitlinie haben wir weiter oben das *Kontextkriterium* eingeführt. Intuitiv – und auch mathematisch – ist das plausibel; wo immer z. B. ganze Zahlen erwartet werden, sind auch natürliche Zahlen willkommen. Aber Programmiersprachen sind rigoroser als Mathematiker; deshalb wird der Zusammenhang präzisiert, indem explizite Operatoren zum *Typtest* und zur *Typanpassung* bereitgestellt werden. Beide haben wir (nicht ganz zufällig) schon im Zusammenhang mit Summentypen kennengelernt (s. Abschnitt 6.5).

Definition (Typtest, Typanpassung (Casting))

Sei S ein Subtyp von T, also $S \subseteq T$. Der ***Typtest*** prüft Werte $t\colon T$ des Supertyps auf ihre Zugehörigkeit zum Subtyp.

t IS S *−− gehört der Wert $t\colon T$ zum Subtyp S?*

Die ***Typanpassung***, auch ***Casting*** genannt, passt den Typ eines gegebenen Wertes an.

s AS T *−− Anpassung von $s\colon S$ an den Supertyp T (Upcast)*
t AS S *−− Anpassung von $t\colon T$ an den Subtyp S (Downcast)*

Die zweite dieser Anpassungen ist nur möglich, wenn (t IS S) gilt. Damit haben wir folgende Operatoren:

FUN _ IS S: $T \to Bool$ *−− Typtest*
FUN _ AS T: $S \to T$ *−− Upcast*
FUN _ AS S: $T \to S$ *−− Downcast (partielle Funktion!)*

Die Casting-Operatoren haben – sofern sie beide definiert sind – folgende Eigenschaften:

$(\_\text{AS } T) \circ (\_\text{AS } S) = id\colon (T \to T)$
$(\_\text{AS } S) \circ (\_\text{AS } T) = id\colon (S \to S)$

Wegen der Reflexivität $S \subseteq S$ gibt es rein formal auch die folgenden beiden Operationen, von denen die erste immer *true* liefert und die zweite die Identität ist:

FUN _IS S: $S \to Bool$ *−− immer true*
FUN _AS S: $S \to S$ *−− Identität*

In nahezu allen Programmiersprachen (sofern sie überhaupt Subtypen bzw. Vererbung kennen) gilt folgende Regel: *Die Anpassung an den Supertyp, also Upcast, wird vom Compiler automatisch vorgenommen.* Sei z. B. eine Funktion FUN $f\colon Int \to \ldots$ gegeben. Dann wird

$\ldots$ LET $n\colon Nat = 2$ IN $f(n)$

automatisch zu folgender Form ergänzt:

$\ldots$ LET $n\colon Nat = 2$ IN $f(n$ AS $Int)$

Da die umgekehrte Richtung, also *Downcast*, potenziell gefährlich ist, bestehen manche Sprachen – z. B. JAVA – darauf, dass der Programmierer die Anpassung selbst schreibt und somit die Verantwortung für das Risiko explizit übernimmt. Wir sehen das allerdings etwas weniger verbissen, sondern erwarten, dass der Compiler zumindest in den wichtigsten Standardsituationen auch die Abwärtsrichtung selbst hinbekommt. Ein typisches Beispiel wäre etwa bei einer Funktion $g\colon Nat \to \ldots$ der Aufruf

$\ldots$ IF x IS Nat THEN $\ldots g(x) \ldots$

Hier kann der Compiler die Anpassung $g(x \text{ AS } Nat)$ problemlos ergänzen.

7.1.3 *Coercion Semantics*

Wie schon erwähnt, macht die Einführung von Subtypen die Typinferenz wesentlich komplexer. Insbesondere wird es durch die im Folgenden eingeführten Konstruktionen unumgänglich, dass wir *dynamische Typprüfung* verwenden. Um ein möglichst einheitliches und durchgängiges System zu erhalten, wählen wir einen Ansatz, der unter dem Begriff *Coercion Semantics* [117] läuft. Dieser Ansatz ist für funktionale Programmierung gut geeignet, weil er die Behandlung von Subtypen letztlich über Anpassungsfunktionen erledigt. Anders ausgedrückt, *wir transformieren eine Sprache mit Subtypen in eine einfachere Sprache ohne Subtypen*, indem wir an geeigneten Stellen entsprechende Anpassungsoperationen einfügen.

Zur Erläuterung betrachten wir ein schematisches Beispiel. Seien zwei Funktionen gegeben:

FUN $f\colon Real \to \ldots$
FUN $g\colon Angle \to \ldots$

wobei $Angle = (Real \mid 0 \leq \_ \leq 360)$ unser bekannter Subtyp der Winkel ist. (Die Notation wird in Abschnitt 7.2 eingeführt.) Seien außerdem zwei Werte gegeben

$r\colon Real = 12.7$
$a\colon Angle = 43.7$

Wenn wir jetzt die beiden Funktionen auf diese Werte anwenden, dann entstehen folgende implizite Castings:

$$\begin{array}{cccc} f(r) & f(a) & g(r) & g(a) \\ \downarrow & \downarrow & \downarrow & \downarrow \\ f(r \text{ AS } Real) & f(a \text{ AS } Real) & g(r \text{ AS } Angle) & g(a \text{ AS } Angle) \end{array}$$

Die erste und die letzte dieser Anpassungen sind die Identität und werden deshalb im Rahmen der Optimierung vom Compiler entfernt. Die zweite Anpassung $f(a \text{ AS } Real)$ ist ein harmloses Upcast, das – je nach Kontext – auch problemlos entfernt werden kann, so dass nur der „blanke“ Wert 43.7 im Code übrig bleibt. Das einzige Problem stellt $g(r \text{ AS } Angle)$ dar, denn hier muss tatsächlich getestet werden, ob r zwischen 0 und 360 liegt.

Um das genauer beschreiben zu können, benötigen wir wieder unsere notationelle Konvention für die Laufzeitdarstellung von Werten und ihren Typen (s. Abschnitt 6.1.3). In dieser Notation können wir jetzt den Effekt der vier Casting-Operationen beschreiben.

$$\begin{array}{lcl} f(r^{\langle Real\rangle}\,\text{AS}\,Real) & = & f(r^{\langle Real\rangle}) \\ f(a^{\langle Angle\rangle}\,\text{AS}\,Real) & = & f(a^{\langle Real\rangle}) \\ g(r^{\langle Real\rangle}\,\text{AS}\,Angle) & = & g(\text{IF } 0 \le r^{\langle Real\rangle} \le 360 \text{ THEN } r^{\langle Angle\rangle} \text{ FI }) \\ g(a^{\langle Angle\rangle}\,\text{AS}\,Angle) & = & g(a^{\langle Angle\rangle}) \end{array}$$

Die beiden Identitäten sind trivial, und das Upcast besteht nur darin, dass der annotierte Typ von a ausgewechselt wird. Nur beim Downcast muss mehr geleistet werden.

Wir haben hier als Entwurfsentscheidung festgelegt, dass beim Casting von Subtypen – anders als beim Summentyp – *nicht* beide Typen annotiert werden. Das heißt, das Casting bewirkt einen echten Informationsverlust. Damit müsste z. B. bei den unmittelbar nacheinander ausgeführten Up- und Downcasts $((a \text{ AS } Real) \text{ AS } Angle)$ der Test $0 \le \_ \le 360$ wieder ausgeführt werden. Diese Entscheidung opfert also Laufzeit für einen Gewinn an Speichereffizienz. (Die umgekehrte Entscheidung wäre aber ebenso möglich.)

Anmerkung: Compilertechnisch sind typannotierte Werte Paare, bestehend aus dem Wert und seinem Typ (in einer geeigneten Codierung). Deshalb ist z. B. $a^{\langle Angle\rangle}$ ein anderes Objekt als $a^{\langle Real\rangle}$ und muss deshalb prinzipiell neu erzeugt und gespeichert werden. Da nach den Castings die verbliebenen Typannotationen aber fast nirgends gebraucht werden, kann der Compiler sie im Rahmen einer Unused-value-*Optimierung meist ebenfalls entfernen.*

7.2 Direkte Subtypen: Constraints

Die direkteste Form, Subtypen einzuführen, besteht darin, einen gegebenen Typ mit einem einschränkenden Prädikat zu versehen.[1]

$$\begin{array}{l} \text{TYPE } Temperature = Real \mid \_ \ge -273 \\ \text{TYPE } Angle = Real \mid 0 \le \_ \le 360 \\ \text{TYPE } Dist = Real \mid \_ \ge 0 \end{array}$$

Aufgrund des Kontextkriteriums gelten in diesen Beispielen offensichtlich die Subtyp-Relationen $Temperature \subseteq Real$, $Angle \subseteq Real$ und $Dist \subseteq Real$.

Das Constraint ist im Allgemeinen ein λ-Ausdruck, den wir allerdings häufig – wie in den obigen Beispielen – mit Hilfe der Wildcard-Notation schreiben können. In voller λ-Notation sieht das z. B. folgendermaßen aus:

$$\text{TYPE } Angle = Real \mid \lambda x \bullet 0 \le x \le 360 \quad -- \textit{ Standardnotation}$$

[1] Wir verwenden den senkrechten Strich „|“ sowohl für die Bildung von direkten Subtypen als auch für die Bildung von Summentypen. Aufgrund des jeweiligen Kontexts ist aber immer klar, was gemeint ist.

Auf den ersten Blick bietet sich noch eine weitere Kurzschreibweise an, die an Schreibweisen der Mengenlehre angelehnt ist:

TYPE $Angle = x{:}\,Real \mid 0 \leq x \leq 360$ $\quad --$ *Kurznotation (gefährlich)*

Diese Notation birgt aber die Gefahr von Mehrdeutigkeiten. Spätestens in Kapitel 8, wenn wir Typen mit ihren Typklassen annotieren können, wäre bei der obigen Notation nicht a priori klar, was hier der Typ ist: x oder *Real*. Man kann natürlich auch optimistisch sein und dem Compiler zutrauen, dass er anhand des Kontexts diese Mehrdeutigkeiten auflöst.

Ein nützlicher Spezialfall der Subtyp-Bildung sind *Intervalle*. Sie spielen eine wichtige Rolle z. B. bei Indextypen für Vektoren und Matrizen. Erfreulicherweise können wir hier eine sehr kompakte Darstellung angeben, ohne neue syntaktische Konstrukte einführen zu müssen. Mit der Funktion

FUN $\_\,..\,\_ : Int \times Int \to (Int \to Bool)$
DEF $(a..b)(i) = a \leq i \leq b$

lassen sich Intervalltypen sehr kompakt schreiben:

TYPE $Index = Int \mid 0..50$ $\quad --$ *Kurzform*
TYPE $Index = Int \mid \lambda\, i \bullet 0 \leq i \leq 50$ $\quad --$ *gleichwertige Langform*

Wir werden (im Zusammenhang mit Typklassen) Intervalltypen noch genauer behandeln.

Definition (direkter Subtyp)

Ein ***direkter Subtyp*** S besteht aus einem *Basistyp* T und einem einschränkenden Prädikat – also einem Booleschen Ausdruck – p, das wir als *Constraint* bezeichnen:

TYPE $S = (T \mid p)$ $\quad --$ *direkter Subtyp,* $S \subseteq T$

Die zu direkten Subtypen gehörigen Test- und Anpassungsoperationen haben wir bereits in Abschnitt 7.1.3 analysiert.

Wenn wir diese Konstruktion mit Hilfe der Klasse *Type* als expliziten Operator einführen wollen, dann müssen wir das folgendermaßen schreiben:

KIND $\_ \mid \_ : \widehat{\alpha}{:}\ \mathit{Type} \times (\alpha \to Bool) \to \mathit{Type}$

Dabei stoßen wir aber auf ein notationelles Problem, das wir erst in Kapitel 8 im Zusammenhang mit *abhängigen Typen* genauer betrachten werden. Es gibt hier eine Abhängigkeit zwischen dem ersten und dem zweiten Argument: Das erste Argument ist ein (beliebiger) Typ, und das zweite eine Funktion mit diesem Typ als Definitionsbereich. Mit der Notation $\widehat{\alpha}$ wird deshalb für das erste Argument ein Name eingeführt, mit dessen Hilfe man das zweite Argument entsprechend präzisieren kann.

Zu beachten ist, dass die direkten Subtypen *keinen Schutz vor ungewollten Anpassungen* liefern. Betrachten wir dazu folgende Situation:

VAL a: $Angle = 180$
FUN f: $Dist \to \ldots$

Ein Aufruf $f(a)$ hat jetzt aufgrund der verfügbaren Casting-Operatoren folgenden Effekt (wobei wir wieder die Notation für die Laufzeitrepräsentation verwenden):

$$\begin{aligned} f(a^{\langle Angle \rangle}\text{AS } Dist) &= f((a^{\langle Angle \rangle}\text{AS } Real)\ \text{AS } Dist) \\ &= f(a^{\langle Real \rangle}\text{AS } Dist) \\ &= f(\text{IF } a^{\langle Real \rangle} \geq 0 \text{ THEN } a^{\langle Dist \rangle} \text{ FI }) \end{aligned}$$

Damit haben wir den Winkel 180° in die Distanz 180 m verwandelt. Das ist sicher nicht beabsichtigt. Aber hier eine Abschottung zu erreichen, ist nicht Zweck des Subtyp-Mechanismus. Dazu dienen andere Mechanismen, die auf Produkt und Summe basieren.

Bei den direkten Subtypen sieht man sehr klar die Unterschiede zwischen *intensionaler* und *extensionaler* Sicht. Wenn wir den obigen Typ

TYPE $Temperature = Real \mid \_ \geq -273$

betrachten, dann ist er in intensionaler Sicht ein Paar bestehend aus dem Typ *Real* und dem Prädikat $(\lambda x \bullet x \geq -273)$; diese Sicht kann zur Laufzeit von den Reflection-Mechanismen der Sprache genutzt werden. In extensionaler Sicht dagegen steht der Typ für die (mathematische) Menge $\{\ t \in \mathbb{R} \mid t \geq -273\ \}$.

7.3 Gruppen/Tupel und Subtypen

Es ist plausibel, dass Gruppentypen in der Subtyp-Relation stehen, wenn diese Beziehung komponentenweise gilt. Aber es mag überraschen, dass der Subtyp *mehr Komponenten* haben kann als der Supertyp.

Definition (Subtyp-Relation für Gruppentypen)

Für Gruppentypen gilt die Subtyp-Relation $S \subseteq T$, wenn S und T folgende Form haben:

TYPE $S = \{\ x_1 = S_1, \ldots, x_k = S_k, x_{k+1} = S_{k+1}, \ldots x_n = S_n\ \}$ −− *Subtyp*
TYPE $T = \{\ x_1 = T_1, \ldots, x_k = T_k\ \}$ −− *Supertyp*

mit $S_i \subseteq T_i$ für $1 \leq i \leq k$.

Diese Festlegung ist kompatibel mit der generellen Idee der Subtypen, wie sie im Kontextkriterium festgelegt ist. Denn in jedem Kontext $c[\ldots]$, in dem ein Element t: T erwartet wird, kann ein Element s: S angeboten werden. Die

einzigen Zugriffe auf t können nämlich über Selektionen $t.x_i$ mit $0 \leq i \leq k$ erfolgen, und die sind auch für s definiert.

Auch mit der Festlegung, dass Subtypen Teilmengen modulo homomorpher Einbettung sind, ist diese Definition verträglich. Wie Abbildung 7.1 (aus extensionaler Sicht) veranschaulicht, umfasst T alle Gruppen t, die aus Funktionen $x_1(t) = a_1, \ldots, x_k(t) = a_k$ bestehen, und zwar für alle möglichen Werte a_i (vgl. Kapitel 4). Dementsprechend steht S für alle Gruppen s, die aus

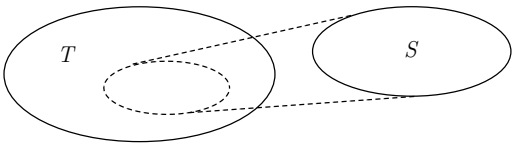


Abb. 7.1: Homomorphe Einbettung bei Gruppen- und Tupeltypen

Funktionen $x_1(s) = a_1, \ldots, x_k(s) = a_k, \ldots, x_n(s) = a_n$ bestehen. Damit lassen sich alle Elemente s, die in $a_1, \ldots, a_k$ übereinstimmen, jeweils auf das entsprechende Element t abbilden.

Der *Upcast* $S \rightarrow T$ ist ganz einfach: Man ignoriert die überflüssigen Komponenten und passt die verbliebenen an.

$$(s \text{ AS } T) = \{\, x_1 = (s.x_1 \text{ AS } T_1), \ldots, x_k = (s.x_k \text{ AS } T_k) \,\}$$

Der *Downcast* $T \rightarrow S$ kann dagegen *nicht* automatisch erzeugt werden.

$$(t \text{ AS } S) = \{\, x_1 = (t.x_1 \text{ AS } S_1),\ \ldots,\ x_k = (t.x_k \text{ AS } S_k),\ x_{k+1} = ?,\ \ldots,\ x_n = ? \,\}$$

denn es ist im Allgemeinen unklar, mit welchen Werten die fehlenden Komponenten belegt werden sollen. Dieses Problem kann nur in folgenden Situationen gelöst werden:

- Beide Typen haben gleich viele Komponenten. Dann kann zumindest ein komponentenweiser Downcast versucht werden.
- Die zusätzlichen Komponenten haben einelementige Typen (wie z. B. *Empty*).
- Bei der Definition des Typs S wurden für die Komponenten Default-Werte vorgesehen. (Ein solches Sprachkonstrukt haben wir hier aber nicht eingeführt.)
- Der Benutzer definiert die Casting-Operation selbst und gibt dabei entsprechende Werte an. Allerdings haben wir hier keine Notationen eingeführt, mit denen sich der Operator _AS_ umdefinieren ließe.

Anmerkung: Das Hinzufügen zusätzlicher Komponenten ist genau die Idee der Vererbung in objektorientierten Sprachen. Dashalb hat Niklaus Wirth dieses Feature in seiner Sprache OBERON *genauso realisiert.*

7.3.1 Subtypen von Tupeltypen

Prinzipiell kann man dieses Design von den Gruppen- auf die Tupeltypen übertragen, denn auch hier kann man nur mit den Selektoren auf die Komponenten zugreifen. Trotzdem erscheint es pragmatisch ratsam, hier auf die Erweiterung der Komponentenzahl zu verzichten. In einer Situation wie

$\text{FUN } f\colon \mathit{Real} \times \mathit{String} \to \ldots$

ist ein Aufruf wie

$\ldots f(3.7, \text{"}\mathit{Volt}\text{"}, 2)\ldots$

vermutlich kein erwünschtes Subtyping, sondern schlicht ein Programmfehler. Und derartige Fehler sollten nicht wegen eines exotischen Subtyp-Features unerkannt bleiben.

Festlegung (Subtypen für Tupeltypen)

Bei Tupeltypen gilt die Subtyp-Relation $S \subseteq T$ genau dann, wenn beide Typen die gleichen Selektoren haben und komponentenweise in Subtyp-Relation stehen.

7.3.2 Produkttypen mit Constraints

Wenn man Subtypen von Gruppen- oder Tupeltypen bilden will, dann kann man natürlich die volle λ-Notation verwenden. Aus Gründen der Lesbarkeit bieten sich hier aber Kurznotationen an, die die ohnehin vorhandenen Selektoren einbeziehen. Wir betrachten als Beispiel wieder unseren bevorzugten Typ *Point*. Unsere bisherigen Variationen haben nämlich ein Problem ignoriert: Alle Punkte im $\mathbb{R}^2$ haben eine eindeutige Darstellung – bis auf den Nullpunkt. Hier ist $\mathit{dist} = 0$, aber der Winkel kann beliebig sein. Folglich sollten wir hier eine Standarddarstellung auszeichnen; dafür bietet sich natürlich $(0, 0)$ an. Das lässt sich folgendermaßen definieren:

$$\text{TYPE } \mathit{Point} = \{\mathit{dist} = \mathit{Dist},\ \mathit{angle} = \mathit{Angle}\} \mid \lambda\, p \bullet p.\mathit{dist} = 0 \implies p.\mathit{angle} = 0$$

Um diese Notation etwas kompakter zu fassen, könnte man sich wieder an mathematischen Gepflogenheiten orientieren und schreiben:

$$\text{TYPE } \mathit{Point} = \{\ \mathit{dist} = \mathit{Dist},\ \mathit{angle} = \mathit{Angle} \mid \mathit{dist} = 0 \implies \mathit{angle} = 0\ \}$$

Allerdings führt auch das wieder auf subtile notationelle Probleme und Mehrdeutigkeiten. Wir werden diese Fragen grundsätzlicher in Kapitel 8 im Zusammenhang mit *abhängigen Typen* behandeln.

7.4 Summentypen und Subtypen

Bei Summentypen erwartet man zu Recht, dass zwischen den einzelnen Varianten und dem Summentyp eine Subtyp-Beziehung herrscht. Das folgt unmittelbar aus dem Kontextkriterium.

Festlegung (Varianten sind Subtypen)

Bei einem Summentyp

$\text{TYPE } T = T_1 \mid \ldots \mid T_n$

gilt für die Varianten die Subtyp-Relation $T_i \subseteq T$.

Die zugehörigen Test- und Anpassungsoperationen wurden schon in Abschnitt 6.5 eingeführt. Die dortige Diskussion zeigt auch, dass man Subtypen als Teilmengen modulo homomorpher Einbettung ansehen muss.

Analog zu Gruppentypen hat man zwischen zwei Summentypen eine Subtyp-Relationen, wenn das für alle Varianten gilt. Bei der Verallgemeinerung auf unterschiedliche Variantenzahl kehrt sich die Situation aber um.

Definition (Subtyp-Relation für Summentypen)

Für Summentypen gilt die Subtyp-Relation $S \subseteq T$, wenn S und T folgende Form haben:

$$\begin{array}{ll} \text{TYPE } S = S_1 \mid \ldots \mid S_k & -- \textit{ Subtyp} \\ \text{TYPE } T = T_1 \mid \ldots \mid T_k \mid T_{k+1} \mid \ldots \mid T_n & -- \textit{ Supertyp} \end{array}$$

mit $S_i \subseteq T_i$ für $1 \leq i \leq k$.

Im Gegensatz zu Gruppentypen hat der Subtyp jetzt *weniger Varianten*. Das ist auch kompatibel mit unseren generellen Anforderungen an Subtypen: In jedem Kontext $c[\ldots]$, in dem ein Element $t\colon T$ erwartet wird, kann ein Element $s\colon S$ angeboten werden. Denn die möglichen Elemente $s\colon S_i$ haben die Eigenschaft $s\colon T_i$ und damit auch $s\colon T$.

Der *Upcast* $S \to T$ ist wieder einfach: Sei $s\colon S_i$ ein Element der Variante S_i, also $(s \text{ IS } S_i)$, dann gilt auch

$$(s \text{ AS } T) = (((s \text{ AS } S_i) \text{ AS } T_i) \text{ AS } T)$$

Der *Downcast* $T \to S$ kann dagegen problematisch werden. Sei $t\colon T$ gegeben mit $(t \text{ IS } T_i)$. Dann gilt für $0 \leq i \leq k$

$$(t \text{ AS } S) = (((t \text{ AS } T_i) \text{ AS } S_i) \text{ AS } S).$$

Dies verlangt, dass der Downcast $T_i \to S_i$ definiert ist. Für die Varianten $T_{k+1}, \ldots, T_n$ ist der Downcast nach S dagegen generell nicht möglich.

Anmerkung: Falls eine Variante S_i von S Subtyp von zwei Varianten T_j und $T_{j'}$ von T ist, dann ist der Upcast mehrdeutig. In diesem Fall muß der Compiler durch eine Typannotation einen Hinweis bekommen, welche Variante gewählt werden soll.

7.4.1 Varianten und „echte“ Subtypen

Wir haben festgestellt, dass Varianten Subtypen ihres Summentyps sind. Das wirft die interessante Frage auf, wie sie sich zu den entsprechenden „echten“ Subtypen verhalten. Wir studieren diesen Zusammenhang anhand unseres Standardbeispiels

TYPE *Shape* = *Point* | *Line* | *Circle* | *Triangle* | *Rectangle*

Wie verhält sich die Varianten-Relation *Point* $\subseteq$ *Shape* zu folgendem direkten Subtyp von *Shape*?

TYPE *Point′* = *Shape* | _ IS *Point*

Was passiert hier? Zuerst bilden wir aus den einzelnen Typen *Point*, ..., *Rectangle* durch disjunkte Vereinigung den neuen Typ *Shape*. Dann extrahieren wir aus diesem großen Typ mittels des Prädikats (_ IS *Point*) denjenigen Subtyp, der gerade dem ursprünglichen Typ *Point* entspricht.

Betrachten wir dazu folgende, leicht skurrile Situation:

FUN f: *Point* $\to \ldots$
FUN f': *Point′* $\to \ldots$
DEF $f'(x') = \ldots f(x') \ldots$

Aufgrund der Typisierung müssen im Rumpf entsprechende Castings eingebaut werden:

DEF $f'(x') = \ldots f(x'$ AS *Shape* AS *Point*$) \ldots$

Um die Komplikation noch zu erhöhen, rufen wir die Funktion f' mit einem Wert p: *Point* auf, also $f'(p)$. Das erzwingt im Argumentausdruck die weiteren Anpassungen $f'(p$ AS *Shape* AS *Point′*$)$. Die Auswertung dieses Aufrufs führt im Rumpf dann wegen der Kompositionseigenschaften der Casting-Operationen insgesamt zu folgender Situation

$$\ldots f(p \text{ AS } \mathit{Shape} \text{ AS } \mathit{Point}' \text{ AS } \mathit{Shape} \text{ AS } \mathit{Point}) \ldots$$

Da sowohl (_ AS *Point′*) $\circ$ (_ AS *Shape*) als auch (_ AS *Shape*) $\circ$ (_ AS *Point*) die Identität sind, vereinfacht sich das zu

$$= \ \ldots f(p \text{ AS } \mathit{Shape} \text{ AS } \mathit{Point}) \ldots$$
$$= \ \ldots f(p) \ldots$$

Es ist illustrativ, sich dieses Beispiel auch in der Laufzeitdarstellung der annotierten Werte anzusehen. Der Aufruf hat die Form

$$\begin{aligned} & f'(p^{\langle \mathit{Point} \rangle} \text{ AS } \mathit{Shape} \text{ AS } \mathit{Point}') \\ = \ & f'(p^{\langle \mathit{Point} \rangle \langle \mathit{Shape} \rangle} \text{ AS } \mathit{Point}') \\ = \ & f'(p^{\langle \mathit{Point} \rangle \langle \mathit{Point}' \rangle}) \end{aligned}$$

Der Aufruf von f im Rumpf führt damit auf folgende Situation:

$$\begin{aligned} & \ldots f(p^{\langle Point\rangle\langle Point'\rangle} \text{ AS } Shape \text{ AS } Point)\ldots \\ = \; & \ldots f(p^{\langle Point\rangle\langle Shape\rangle} \text{ AS } Point)\ldots \\ = \; & \ldots f(p^{\langle Point\rangle})\ldots \end{aligned}$$

7.4.2 Summen + Tupel sind ein Schutzwall

Ein jahrzehntealter Disput im Zusammenhang mit Summentypen ist die Frage, ob sie hinreichend viel Schutz gegen fehlerhafte Programme bieten. Das klassische Beispiel lässt sich folgendermaßen skizzieren:

TYPE $Euro = Int$
TYPE $Dollar = Int$

Kann man hier versehentlich Euro in Dollar umwandeln oder Euro mit Dollar addieren? Betrachten wir dazu die beiden Deklarationen $e\colon Euro = 100$ und $d\colon Dollar = e$. Das führt zu folgenden Anpassungen

$$\begin{aligned} \text{VAL } d\colon Dollar \; & = e^{\langle Euro\rangle} \text{AS } Dollar \\ & = e^{\langle Euro\rangle} \text{AS } Int \text{ AS } Dollar \\ & = e^{\langle Int\rangle} \text{AS } Dollar \\ & = e^{\langle Dollar\rangle} \end{aligned}$$

In dieser naiven Form ist die falsche Konversion offensichtlich möglich. Das ist auch kompatibel mit unseren Intentionen, denn Typsynonyme sollen gerade keine Zwischenschichten einziehen.

Wenn wir einen echten Schutz vor falschen Castings etablieren wollen, müssen wir beide Typen zu einelementigen Produkten machen, wobei wir die Variante mit Konstruktor und Selektor wählen.

TYPE $Euro = euro(\, value = Int\,)$
TYPE $Dollar = dollar(\, value = Int\,)$

Damit ist das naive Casting über Int, das wir oben benutzt haben, ausgeschlossen. Und Tupel sind höchstens dann in Subtyp-Relation, wenn sie gleiche Konstruktoren und Selektoren haben.

Aber es gibt noch eine potenziell gefährliche Situation, die zu hinterfragen ist. Nehmen wir an, wir betten die beiden Typen in einen gemeinsamen Summentyp ein.

TYPE $Currency = Euro \mid Dollar$

Für diesen Typ sei außerdem eine Addition programmiert.

FUN $add\colon Currency \times Currency \to Currency$
DEF $add(x, y) =$
  IF x IS $Euro \wedge y$ IS $Euro$ THEN $euro(x.value + y.value)$
  IF x IS $Dollar \wedge y$ IS $Dollar$ THEN $dollar(x.value + y.value)$ FI

Hier beweisen wir schon großes Vertrauen in die deduktiven Fähigkeiten unseres Compilers. Denn er muss z. B. aus der Bedingung x IS $Euro$ schließen,

dass im entsprechenden THEN-Zweig der Downcast (x AS *Euro*) zu nehmen ist, damit die Applikation $x.value$ definiert ist. (Ein weniger cleverer Compiler wird melden, dass er sich nicht zwischen (x AS *Euro*) und (x AS *Dollar*) entscheiden kann, weil beide eine Operation *value* besitzen.)

Was geschieht, wenn wir versuchen, die beiden Werte $e\colon Euro = 100$ und $d\colon Dollar = 200$ zu addieren?

$$\begin{aligned} add(e^{\langle Euro\rangle},\, d^{\langle Dollar\rangle}) &= add(e^{\langle Euro\rangle}\,\text{AS}\; Currency,\, d^{\langle Dollar\rangle}\,\text{AS}\; Currency) \\ &= add(e^{\langle Euro\rangle\langle Currency\rangle},\, d^{\langle Dollar\rangle\langle Currency\rangle}) \end{aligned}$$

Hier sind beide Bedingungen der Fallunterscheidung von *add* verletzt und die Funktion ist daher undefiniert. Das heißt, die Einbettung in den Summentyp *Currency* hebt die Absicherung über die einelementigen Tupel *nicht* auf.

Übrigens sollte man als Ergebnistyp *Maybe Currency* (s. Abschnitt 8.1.1) wählen und einen entsprechenden ELSE-Zweig hinzuzufügen. Dann wird der Programmabsturz durch eine geordnete Fehlerbehandlung ersetzt.

7.5 Funktionstypen und Subtypen

Beim Verhältnis der Subtyp-Relation zu Funktionstypen zeigt sich ein unangenehmer, aber unvermeidbarer Effekt. Betrachten wir dazu die Funktionen

FUN $f\colon Int \to Real$
FUN $g\colon Nat \to Complex$

wobei wir die üblichen Subtyp-Beziehungen $Nat \subseteq Int$ und $Real \subseteq Complex$ voraussetzen. Dann können wir f überall verwenden, wo g erwartet wird.

FUN $h\colon (Nat \to Complex) \to \ldots$
DEF $h(g) = \;\ldots\; c\colon Complex = g(n\colon Nat)\;\ldots$

Dann können wir problemlos $h(f)$ aufrufen. Denn das führt bei der Auswertung im Rumpf zu folgender Situation

$h(f) \rightsquigarrow \ldots\, c\colon Complex = (f(n \text{ AS } Int)) \text{ AS } Complex \ldots$

Beide Anpassungen sind harmlose Upcasts. Damit liefert das Kontextkriterium hier folgende Subtyp-Relation zwischen den beiden Funktionstypen:

$(Int \to Real) \subseteq (Nat \to Complex)$

Definition (Funktionstypen und Subtyp-Relation)

Die Subtyp-Relation für Funktionstypen ist *contravariant* im Argument und *covariant* im Resultat: Sei $S_1 \subseteq T_1$ und $S_2 \subseteq T_2$; dann gilt

$(T_1 \to S_2) \subseteq (S_1 \to T_2)$

8

Polymorphe und abhängige Typen

Was will meine Einfalt bei ihrer Vielfalt!
Nietzsche (Zarathustra)

Viele Typkonstruktionen erfolgen völlig analog. So sind z. B. Sequenzen von reellen Zahlen auch nicht anders aufgebaut als Sequenzen von Buchstaben oder Sequenzen von Bankkunden. Und Arrays der Länge 100 unterscheiden sich nicht wesentlich von Arrays der Länge 200. Das ist die gleiche Erkenntnis, die man von Funktionen kennt: Weil die Berechnung des Sinus von 32° auch nicht anders erfolgt als die des Sinus von 49°, führt man eine Funktion $sin(x)$ ein, die ein generelles Verfahren für beliebige Werte x definiert.

Dieser Trick funktioniert nicht nur bei Werten, sondern auch bei Typen. Allerdings hat es lange gedauert, bis man sich in den Programmiersprachen dazu durchringen konnte, diese Analogie zu nutzen. Der Grund liegt in der traditionellen Verwendung von Typen: Sie sollten zur Compilezeit Fehler erkennen lassen, und zwar effizient und ohne selbst neue Berechnungsprobleme zu kreieren. Mit dieser Restriktion waren die Möglichkeiten ziemlich begrenzt. Das Fortschrittlichste, was sich in der Praxis durchgesetzt hat, war die ML-artige Polymorphie, die mit dem so genannten Hindley-Milner-Algorithmus im Compiler überprüft werden konnte. In leichten Variationen findet sich das Konzept unter Namen wie *Templates* (in C++) oder *generische Typen*, *generische Klassen*, *parametrische Polymorphie* etc. In der Sprache JAVA haben die Designer bis zur Version 5.0 gebraucht, bevor sie sich zur Aufnahme der Polymorphie durchringen konnten.

Die Komplexität der Situation lässt sich noch steigern. In den so genannten *abhängigen Typen* (engl.: *dependent types*) werden Typen in Abhängigkeit von Werten bestimmt. Damit werden die Grenzen zwischen der Welt der Werte und der Welt der Typen diffus – mit den entsprechenden Komplikationen für Analysealgorithmen.

Wir werden das Thema hier in zwei Schritten behandeln:

- Zuerst betrachten wir die klassische Polymorphie, also Funktionen, die aus *Typen* neue *Typen* generieren.
- Dann wenden wir uns den abhängigen Typen zu, also Funktionen, die aus *Werten* neue *Typen* generieren.

Das im Folgenden skizzierte Konzept basiert im Wesentlichen auf Techniken, die dem so genannten ML-Polymorphismus zugrunde liegen und von dort aus den Weg in viele Programmiersprachen gefunden haben. Wir wollen aber in der Ausdrucksmächtigkeit weiter gehen und müssen deshalb diese ML-artige Notation in ein allgemeineres Konzept einbetten. Im Gegensatz zur klassischen Typtheorie, in der das Thema primär in der Form von Typ*termen* und deren Manipulation betrachtet wird, werden wir eine stärker *funktional* orientierte Darstellungsform wählen, die sich homogener in den Rest des Sprachdesigns einfügt.

In der Typtheorie gibt es natürlich zahlreiche Untersuchungen zu diesem Fragenkomplex. Diese bauen meist auf dem so genannten *System F* auf, das im Wesentlichen von Girard [57] und ähnlich auch von Reynolds [124] eingeführt wurde. Genaueres kann man wieder in [117, 118] oder [30] nachlesen.

8.1 Typfunktionen (generische Typen, Polymorphie)

Wie in der Einleitung des Kapitels schon erwähnt, gibt es keinen Grund, die Idee der Funktionen nicht auch auf Typen auszudehnen. Wegen seiner Komplexität nähern wir uns dem Thema in mehreren Schritten.

8.1.1 Typvariablen

Die Erweiterung auf polymorphe und abhängige Typen führt eine neue Komplexität in die Typterme ein: *Typterme können jetzt freie Variablen enthalten.* Das ist zwar kein grundsätzliches theoretisches Problem, aber es stellt eine Herausforderung für das Sprachdesign dar, weil die Lesbarkeit nicht allzu sehr leiden darf. Für den Augenblick lösen wir dieses Problem ganz pragmatisch, indem die Typvariablen durch die Verwendung griechischer Buchstaben kennzeichnen.

Beispiel 1. Paare über Elementen beliebiger Typen lassen sich folgendermaßen definieren:

TYPE $Pair\ \alpha\ \beta = (1st = \alpha,\ 2nd = \beta)$

Anstelle der Notation mit Currying hätte man auch eine Definition in Tupelschreibweise wählen können:

TYPE $Pair'(\alpha, \beta) = (1st = \alpha,\ 2nd = \beta)$ $\quad --\ Tupelnotation$

Beispiel 2. Oft hat man es mit Funktionen zu tun, die normalerweise ein Resultat von einem gewissen Typ liefern, aber unter Umständen auch auf einen

Fehler laufen können. Für die reellen Zahlen ist dafür im IEEE-Standard explizit eine Vorkehrung getroffen worden, die wir in Abschnitt 6.5.3 vorgestellt haben. Aber das gleiche Problem tritt bei allen möglichen Typen auf. Für diese Situationen ist folgender Typ nützlich:

TYPE *Maybe* $\alpha = \alpha \mid \mathit{Fail}$
TYPE *Fail* $= \{\, \mathit{fail} \,\}$

Beispiel 3. Am wichtigsten ist dieses Konzept im Zusammenhang mit rekursiven Typen. Das Standardbeispiel der Listen über beliebigen Elementtypen haben wir bereits kennen gelernt.

TYPE *List* α = $(\mathit{ft} = \alpha,\ \mathit{rt} = \mathit{List}\ \alpha)$
| *Empty*
TYPE *Empty* $= \{\, \Diamond \,\}$

Die *Instanziierung* solcher Typterme mit freien Variablen erfolgt analog zur Schreibweise bei normalen Funktionen: Anstelle der Typvariablen wird ein konkreter Typ eingesetzt. Um der Klarheit willen sollten gegebenenfalls Klammern benutzt werden.

List Nat	bzw.	*List*(*Nat*)
List (*Pair Nat String*)	bzw.	*List*((*Pair Nat*) *String*)
List Pair′(*Nat*, *String*)	bzw.	*List*(*Pair′*(*Nat*, *String*))

Auch die Frage der *Verwendung* solcher Typterme muss geklärt werden. Denn das Einführen eines generischen Typs ist kein Selbstzweck. Wie alle Typen dient er vor allem dazu, die Definitions- und Wertebereiche von Funktionen zu beschreiben. Betrachten wir ein klassisches Beispiel:

FUN *length*: *List* $\alpha \to$ *Nat* $\quad --$ *problematische Notation*
DEF *length* $s = \ldots$

Das Problem hier ist das freie Vorkommen der Typvariablen α, die ja nur aufgrund unserer Konvention der Verwendung griechischer Buchstaben als solche zu erkennen ist.

Wenn die Funktion *length* auf eine konkrete Liste angewandt wird, muss der Compiler durch eine so genannte *Unifikation* analysieren, wofür α steht und ob die daraus resultierende konkrete Typisierung korrekt ist. Sei z. B. eine Liste *pl*: *List* (*Pair Nat Real*) gegeben, dann wird beim Aufruf *length*(*pl*) die Typvariable α mit dem konkreten Typ (*Pair Nat Real*) instanziiert.

Die Verwendung freier Variablen liefert sehr bequeme Schreibweisen, führt aber auf eine Reihe subtiler technischer und notationeller Probleme.

- Das erste Problem ist die *syntaktische Charakterisierung* der Typvariablen. In der Literatur findet man hierzu eine Vielzahl von Notationen, insbesondere Πx, $\forall x$, λx, Λx oder auch $\widehat{x}$.

- Das nächste Problem ist der *Bindungsbereich*. Es muss syntaktisch klar sein, auf welchen (Sub-)Term sich die Variable bezieht. Dies kann insbesondere im Zusammenhang mit Subtypen diffizil werden. Wir werden auch auf notationelle Mehrdeutigkeiten stoßen, sobald wir (in Kapitel 9) das Konzept der Typklassen einführen. (Eine detaillierte Analyse dieser vielfältigen Probleme kann man in [117] und [118] nachlesen.)
- Schließlich muss man auch die *adäquate Instanziierung* sicherstellen: So muss z. B. bei mehreren Anwendungen von *length* auf Listen unterschiedlicher Typen die freie Variable α jeweils neu instanziiert werden.

Wir gehen auf die Fragen einer adäquaten Notation erst am Ende dieses Kapitels in Abschnitt 8.3 ein. Bis dahin behelfen wir uns mit folgender Verabredung:[1]

Festlegung (Variablen in Typtermen)

In einfachen Fällen kennzeichnen wir freie Typvariablen durch die Verwendung griechischer Buchstaben (wie z. B. in *Seq* α).

Anmerkung: Die Verwendung von Spezialnotationen für Typvariablen hat Tradition. So schreibt man z. B. in ML `'a list` *für unser List* α. *In* HASKELL *ist man hier moderner und erlaubt beliebige Identifier, die – wie alle Variablennamen in* HASKELL *– klein geschrieben werden, also z. B.* `[a]`.

8.1.2 Typfunktionen

Jedes Konzept, das mit freien Variablen arbeitet, führt früher oder später auf subtile Probleme mit den Bindungsbereichen. Aus diesem Grund verwendet man in normalen Programmtermen λ-Ausdrücke. Denn dadurch gibt es keine freien Variablen mehr und die Bindungsbereiche sind grundsätzlich wohl definiert. Wenn wir dieses Prinzip auf polymorphe Funktionen anwenden, dann ergeben sich Notationen der folgenden Bauart:

DEF $id = \lambda\alpha \bullet \lambda x{:}\,\alpha \bullet x$
DEF $twice = \lambda\alpha \bullet \lambda f{:}\,(\alpha \to \alpha) \bullet \lambda x{:}\,\alpha \bullet f(f(x))$
DEF $length = \lambda\alpha \bullet \lambda s{:}\,List\ \alpha \bullet$ IF s IS $Empty$ THEN 0
ELSE $1 + length(\alpha)(rt\ s)$ FI

Bei der *Anwendung* solcher Funktionen müssen wir sowohl die normalen Argumentwerte als auch die Instanz des Typparameters angeben:

$\ldots id(Nat)(5)\ldots$
$\ldots twice(Real)(sqrt)(3)\ldots$
$\ldots length(Char)(text)\ldots$

[1] Im Interesse der Lesbarkeit werden wir diese notationelle Verabredung im ganzen Buch benutzen.

Aber dabei bleibt eine Frage offen: *Was sind die Typen dieser Funktionen?* Als Zwischenschritt betrachten wir die Definitionen nochmals, aber jetzt in einer (teilweise) musterbasierten Form:

DEF $id(\alpha) = \lambda x{:}\,\alpha \bullet x$
DEF $twice(\alpha) = \lambda f{:}\,(\alpha \to \alpha) \bullet \lambda x{:}\,\alpha \bullet f(f(x))$
DEF $length(\alpha) = \lambda s{:}\,List\ \alpha \bullet$ IF s IS $Empty$ THEN 0
ELSE $1 + length(\alpha)(rt\ s)$ FI

Für die so teilinstanziierten Funktionen lassen sich die Typen ganz normal angeben (was unser eigentliches Problem aber noch nicht löst):

FUN $id(\alpha)$: $\alpha \to \alpha$
FUN $twice(\alpha)$: $(\alpha \to \alpha) \to \alpha \to \alpha$
FUN $length(\alpha)$: $List\ \alpha \to Nat$

Wir wissen jetzt zwar, was der Typ von $id(\alpha)$ oder von $length(\alpha)$ ist, aber noch immer nicht, was der Typ von id bzw. $length$ ist. Um dieses Problem endgültig zu lösen, brauchen wir wieder die Klasse *Type* aller Typen; allerdings reicht das alleine nicht aus, wie man an dem folgenden (unzureichenden) Versuch sieht:

FUN id: $Type \to \alpha \to \alpha$ $\quad$ $--$ *reicht nicht!*

Hier wird zwar korrekt gesagt, dass die Funktion id als erstes Argument einen Typ bekommt und als zweites Argument einen Wert (wegen unserer Verabredung, dass griechische Buchstaben für Typvariablen stehen); aber es fehlt die Information, dass das zweite Argument denjenigen Typ haben soll, der als erstes Argument übergeben wird. Damit sind wir in dem Problemkreis gelandet, der unter dem Stichwort *abhängige Typen* diskutiert wird.

Bevor wir diesen Problemkreis in voller Allgemeinheit diskutieren, wollen wir zumindest noch unser Beispiel zu Ende bringen. Wie wir im weiteren Verlauf dieses Kapitels noch erläutern werden, brauchen wir eine zweite Art von Pfeil, mit der dann folgende Notation möglich wird:[2]

FUN id: $\alpha{:}\,Type \mapsto \alpha \to \alpha$
FUN $twice$: $\alpha{:}\,Type \mapsto (\alpha \to \alpha) \to \alpha \to \alpha$
FUN $length$: $\alpha{:}\,Type \mapsto List\ \alpha \to Nat$

Mit dieser Notation können wir jetzt auch Typdeklarationen sauber aufschreiben. Wir betrachten noch einmal die drei Beispiele des vorigen Abschnitt 8.1.1.

[2] Die Notation lehnt sich an die Tradition der Mengenlehre an, wo mit $A \to B$ die *Menge aller Abbildungen* der Menge A in die Menge B bezeichnet wird. Demgegenüber bezeichnet $a \mapsto b$ eine *einzelne konkrete Abbildung*, die dem Wert a den Wert b zuordnet.

TYPE $Pair = \alpha\colon \mathit{Type} \mapsto \beta\colon \mathit{Type} \mapsto (\mathit{1st} = \alpha,\ \mathit{2nd} = \beta)$
TYPE $Pair' = \alpha\colon \mathit{Type} \times \beta\colon \mathit{Type} \mapsto (\mathit{1st} = \alpha,\ \mathit{2nd} = \beta)$
TYPE $\mathit{Maybe} = \alpha\colon \mathit{Type} \mapsto (\alpha \mid \mathit{Fail})$
TYPE $\mathit{List} = \alpha\colon \mathit{Type} \mapsto (\ (\mathit{ft} = \alpha,\ \mathit{rt} = \mathit{List}\ \alpha\ )\ \mid\ \mathit{Empty}\ )$

Damit können wir jetzt auch unsere ursprüngliche Notkonvention aufgeben, dass Typvariablen immer mit griechischen Buchstaben zu schreiben sind. Jetzt können wir beliebige Identifier nehmen, denn ihre Rolle ist durch die Art des Pfeils eindeutig festgelegt.

Weitere Erläuterungen zur Schreibweise werden wir gleich noch in Abschnitt 8.3 geben. Zuvor wenden wir uns aber dem generellen Thema der abhängigen Typen zu.

8.2 Abhängige Typen

Im vorigen Abschnitt 8.1 haben wir klassische polymorphe Typen betrachtet, also Funktionen, die Typen in Typen abbilden. Die nächste Stufe der Verallgemeinerung sind Funktionen, die *Werte* in *Typen* abbilden. Diese Situation ist unter dem Namen ***abhängige Typen*** bekannt (engl.: *dependent types*). Zur Motivation beginnen wir mit illustrierenden Beispielen.

Beispiel 8.1 (Skalarprodukt)

Das klassische Beispiel sind Arrays. Wenn man z. B. das Skalarprodukt zweier Vektoren berechnen will, dann müssen diese die gleiche Länge haben. Das lässt sich folgendermaßen ausdrücken:

FUN $\mathit{skalProd}\colon\ n\colon \mathit{Nat} \mapsto \mathit{Vector}(n) \times \mathit{Vector}(n) \to \mathit{Real}$
DEF $\mathit{skalProd}(k)(a, b) = \ldots$

Dazu benötigt man eine Möglichkeit, entsprechende Subtypen des generellen Typs *Array* zu definieren.

TYPE $\mathit{Vector} = n\colon \mathit{Nat} \mapsto (\mathit{Array}\ \mathit{Real} \mid \lambda\, a \bullet a.\mathit{length} = n)$

Man beachte, dass man beides auch hätte musterbasiert schreiben dürfen:

FUN $\mathit{skalProd}(n\colon \mathit{Nat})\colon\ \mathit{Vector}(n) \times \mathit{Vector}(n) \to \mathit{Real}$
TYPE $\mathit{Vector}(n\colon \mathit{Nat}) = (\mathit{Array}\ \mathit{Real}\ \mid\ \lambda\, a \bullet a.\mathit{length} = n)$

Beispiel 8.2 (Die Funktion *random*)

Abhängige Typen erhöhen die Ausdrucksmächtigkeit des Typsystems und machen es damit nützlicher. Das zeigt sich sehr schön am Beispiel der Funktion $\mathit{random}(a, b)$ die eine Zufallszahl aus dem Bereich $[a \ldots b]$ liefert. In klassischen Programmiersprachen hat diese Funktion folgenden Typ:

FUN $random\colon Real \times Real \to Real$ $\quad -- \textit{ korrekter, aber ungenauer Typ}$

Das ist aber nur eine schwache Approximation an den wirklichen Typ. Wenn man exakter beschreiben will, welche Resultate bei dieser Funktion zu erwarten sind, muss man einen entsprechenden Subtyp verwenden:

FUN $random\colon a\colon Real \times b\colon Real \mapsto (Real \mid a..b)$ $\quad -- \textit{ genauer Typ}$

Bei aller Eleganz bringt dieses Konzept leider auch enorme Schwierigkeiten mit sich. Das deutet sich in einem weiteren Beispiel noch klarer an.

Beispiel 8.3 (Listen fester Länge)

Wir betrachten Listen einer festen Länge.

TYPE $FixedList = n\colon Nat \mapsto \alpha\colon \mathit{Type} \mapsto (List(\alpha) \mid \lambda l \bullet l.length = n)$

Dann hat eine Operation wie *cons* folgenden Typ (zur Abwechslung in musterbasierter Form geschrieben):

FUN $cons(n)(\alpha)\colon\ \alpha \times FixedList(n)(\alpha) \to FixedList(n+1)(\alpha)$

Das heißt, die Feststellung des Typs verlangt eine explizite Berechnung. Und prinzipiell kann niemand daran gehindert werden, anstelle einer simplen Addition auch komplexe rekursive Funktionen in die Typbestimmung einzubauen.

Wir wollen uns aber nicht nur auf diese kleinen Spielbeispiele zur Illustration beschränken, sondern auch eine praktische Anwendung zeigen.

Beispiel 8.4 (Die Funktion *printf*)

Ein berühmtes Beispiel für die Verwendung abhängiger Typen ist die Funktion `printf` der Sprache C. Diese Funktion hat als erstes Argument einen String, in dem eingestreute Formatzeichen als Platzhalter für die Werte fungieren, die als weitere Argumente angegeben sein müssen. Beispiele:

$printf(\texttt{"}Der\ \%d.\ Name\ ist\ \%s\texttt{"})(1, \texttt{"}Meier\texttt{"})$
$printf(\texttt{"}Herr\ \%s\ zahlt\ \%d\ Euro\ an\ Herrn\ \%s\texttt{"})(\texttt{"}Meier\texttt{"}, 280, \texttt{"}Huber\texttt{"})$

Der Typ der Funktion *printf* hängt also – in nichttrivialer Weise – vom ersten Argument ab. In den beiden obigen Beispielen haben wir die Typen

FUN $printf\colon\ String \to (Int \times String) \to String$
FUN $printf\colon\ String \to (String \times Int \times String) \to String$

Wenn wir die Typisierung von *printf* allgemein beschreiben wollen, brauchen wir relativ komplexe abhängige Typen. Dazu nehmen wir an, dass wir eine normale Funktion

FUN $format$: $String \to List(Char)$
DEF $format(s) = \ldots$

zur Verfügung haben, die aus einem gegebenen String die Liste der Formatzeichen extrahiert, im obigen Beispiel also die Listen $\langle \texttt{"}d\texttt{"}, \texttt{"}s\texttt{"} \rangle$ bzw. $\langle \texttt{"}s\texttt{"}, \texttt{"}d\texttt{"}, \texttt{"}s\texttt{"} \rangle$.

Mit Hilfe dieser Funktion können wir jetzt die Typisierung der Funktion *printf* angeben, wobei die wesentliche Arbeit von einem abhängigen Typ *Tuple* geleistet wird, also einer Funktion, die aus einer Liste von n Werten einen Produkttyp mit n Komponenten macht.

FUN $printf$: $s\colon String \mapsto Tuple(format(s)) \to String$

Die Typfunktion *Tuple* ist rekursiv definiert, wie das bei Funktionen auf Listen üblich ist. Der Unterschied zu normalen Funktionen ist jetzt aber, dass das Ergebnis ein Typ ist.

KIND $Tuple$: $List(Char) \to \mathit{Type}$
DEF $Tuple(\Diamond) = ()$
DEF $Tuple(\texttt{"}d\texttt{"} \cdot\colon rest) = Int \otimes Tuple(rest)$
DEF $Tuple(\texttt{"}s\texttt{"} \cdot\colon rest) = String \otimes Tuple(rest)$
$\ldots$

Man beachte, dass zur Bildung des Produkttyps der Operator „ $\otimes$ “ verwendet wird, weil hier einer der seltenen Fälle vorliegt, wo das assoziative Produkt gebraucht wird.

Die letzten beiden Beispiele zeigen besonders deutlich, dass abhängige Typen in dieser Allgemeinheit höchstens im Rahmen von *dynamischer Typprüfung* behandelbar sind. In den wenigen Sprachen, die sich überhaupt an das Konzept der abhängigen Typen wagen, werden deshalb auch starke Einschränkungen vorgenommen, um eine statische Typprüfung zu ermöglichen (wie z. B. in RUSSEL oder DEPENDENT ML), oder es wird in Kauf genommen, dass der Compiler unter Umständen nicht terminiert (wie in CAYENNE).

Die obigen Beispiele liefern eine intuitive Vorstellung vom Konzept der abhängigen Typen. Allerdings zeigen sie auch, dass die polymorphen Typen und die abhängigen Typen kaum voneinander abgrenzbar sind; insbesondere ist es leicht, Mischfälle zu konstruieren. Deshalb fassen wir beides zusammen:

Definition (abhängiger Typ)

Im allgemeinsten Fall ist ein ***abhängiger Typ*** (engl.: *dependent type*) eine Funktion, die Typen liefert. Das schließt zwei Fälle ein:

- Wenn das Argument ein Typ ist, spricht man von *polymorphen Typen.*
- Wenn das Argument ein Wert ist, spricht man von *abhängigen Typen* (im engeren Sinn).

Als Notation verwenden wir Schreibweisen der Art

$$x\colon A \mapsto B$$

Dabei sind A und B der Definitions- bzw. Wertebereich der Funktion und x die gebundene Variable, mit deren Hilfe die Abhängigkeit formuliert wird. Wenn x nicht in B vorkommt, ist das äquivalent zum normalen Funktionstyp:

$(A \to B)$ ist gleichwertig zu $x\colon A \mapsto B$ falls x nicht in B vorkommt

---

Der neue Pfeil „ $\mapsto$ “ bewirkt eine subtile Änderung der Rollen der beteiligten Symbole. Betrachten wir zur Illustration drei Beispiele, wobei die Unterstreichung jeweils anzeigt, was den Definitions- bzw. Wertebereich angibt.

FUN $f\colon \underline{Int} \to \underline{Real}$ *−− normaler Typ*
FUN $f\colon \underline{Int}\colon Type \to \underline{Real}\colon Type$ *−− normaler Typ mit Annotationen*
FUN $g\colon \alpha\colon \underline{Type} \mapsto \underline{List\ \alpha}$ *−− abhängiger (polymorpher) Typ*
FUN $printf\colon s\colon \underline{String} \mapsto \underline{Tuple(format(s)) \to String}$ *−− abhängiger Typ*

In der ersten Zeile haben wir eine normale Funktion von Int nach $Real$. In der zweiten Zeile haben wir genau die gleiche Funktion, wobei wir jetzt – überflüssigerweise – Int und $Real$ durch eine Annotation als Typen charakterisieren. (Das wird sich in Kapitel 9 ändern; dort bekommen solche Annotationen mit Typklassen eine relevante Bedeutung.) In der dritten und vierten Zeile ändert sich die Situation: durch den Pfeil „ $\mapsto$ “ wird das Symbol vor dem „ : “ als gebundene Variable gekennzeichnet; der Definitionsbereich wird jetzt durch das Symbol hinter dem „ : “ angegeben.

Anmerkung: Mit den abhängigen Typen haben wir ein neues Konzept mit neuen Notationen eingeführt. Die Frage ist, ob das zwingend notwendig ist oder nur bequem. Betrachten wir dazu noch einmal das obige Beispiel der Funktion random. Wir hatten sie mit Hilfe eines abhängigen Typs charakterisiert:

FUN $random\colon a\colon Real \times b\colon Real \mapsto (Real \mid a\,..\,b)$

Die gleiche Information kann mittels Subtypen angegeben werden:

FUN $random\colon ((Real \times Real \to Real) \mid \lambda f \bullet \forall a, b \bullet a \leq f(a,b) \leq b)$

Hier gehen wir von dem normalen Funktionstyp $(Real \times Real \to Real)$ aus und schränken ihn mit einem Prädikat ein, das besagt, dass nur Funktionen zu dem Subtyp gehören, deren Resultat zwischen den beiden Argumenten liegt.

Wie man an diesem Beispiel sieht, ist der Aufwand aber ebenfalls relativ hoch und die Lesbarkeit meistens schlechter. Deshalb werden wir im weiteren Verlauf des Buches lieber mit abhängigen Typen (und später auch Typklassen) arbeiten, als den mühsamen Weg über Subtyp-Bildungen zu wählen.

8.3 Notationen für Typterme mit Variablen

Wie wir in den vorangegangenen Abschnitten gesehen haben, stellen sich durch die Einführung von Variablen in Typtermen drei Fragen:

- Wie kennzeichnet man die entsprechenden Identifier als Variablen?
- Wie legt man den Bindungsbereich fest?
- Wie erhält man eine akzeptable Lesbarkeit?

In der klassischen Typtheorie löst man die ersten beiden Aspekte ganz einfach, indem man Bindungsoperatoren einführt, wobei man den dritten Aspekt schlicht ignoriert. Wenn man sich mit *Sprachdesign* befasst, wird dieser dritte Aspekt jedoch sehr relevant.

8.3.1 Bindung von Variablen

In der Typtheorie wird anstelle unserer Notation $(x\colon A \mapsto B)$ oft die so genannte Π-Notation verwendet: $(\Pi x\colon A \bullet B)$. Dabei erfüllt das Symbol Π die gleiche Funktion wie das Symbol λ bei normalen Funktionen: es bindet die Variable x. Wenn man – z. B. im System F – von *universell quantifizierten Typen* spricht, verwendet man konsequenterweise einen Allquantor. Damit sehen einige unserer Beispieltypen dann folgendermaßen aus:

FUN *id*: $\forall\alpha\colon \mathit{Type} \bullet \alpha \to \alpha$ −− *System F*
FUN *twice*: $\forall\alpha\colon \mathit{Type} \bullet (\alpha \to \alpha) \to \alpha \to \alpha$ −− *System F*
FUN *length*: $\forall\alpha\colon \mathit{Type} \bullet \mathit{List}\ \alpha \to \mathit{Nat}$ −− *System F*

Anmerkung: Das Typsystem F wird impredicative genannt, weil der Quantor in der Definition sich über eine Menge erstreckt, zu der das definierte Element selbst gehört. Um diesen reichlich mystischen Satz zu verstehen, betrachte man z. B. den Typ TYPE $T = \forall\alpha \bullet \alpha \to \alpha$*, der zur Identitätsfunktion gehört, d. h. id*: T*. Sei nun ein Element* t: T *gegeben, dann können wir* $id(t)$ *schreiben; diese Instanz von id hat den konkreten Typ* $T \to T$*. So etwas ist bei* ML*-artiger Polymorphie mit freien Typvariablen nicht möglich, weshalb man diese auch predicative (oder stratified) nennt (vgl. [117]). (Beide Begriffe stammen aus der Logik.)*

Wir haben uns entschlossen, anstelle eines expliziten Bindungsoperators lieber ein anderes Infixsymbol zu verwenden, nämlich „ $\mapsto$ “ statt „ $\to$ “. Der Effekt ist aber der gleiche.

FUN *id*: $\alpha\colon \mathit{Type} \mapsto \alpha \to \alpha$
FUN *twice*: $\alpha\colon \mathit{Type} \mapsto (\alpha \to \alpha) \to \alpha \to \alpha$
FUN *length*: $\alpha\colon \mathit{Type} \mapsto \mathit{List}\ \alpha \to \mathit{Nat}$

Hier wird α als gebundene Variable eingeführt, deren Bindungsbereich der gesamte folgende Typterm ist. Durch die Annotation mit *Type* wird auch festgelegt, welchen Definitionsbereich das entsprechende Argument hat. (Das wird in Kapitel 9 im Zusammenhang mit den so genannten Typklassen wichtig werden.)

8.3.2 *Syntactic sugar*: Nachgestellte Variablen

In unseren bisherigen Beispielen waren die Anzahl und die Annotationen der Variablen jeweils sehr einfach, typischerweise von der Art $n{:}\,Int$ oder $\alpha{:}\,\mathit{Type}$. Das wird sich in späteren Kapiteln ändern (z. B. in Kapitel 14 im Zusammenhang mit Arrays); dort werden wesentlich komplexere Bindungen auftreten, was neue notationelle Herausforderungen mit sich bringt.

In normalen Wertausdrücken verwendet man LET- oder WHERE-Klauseln dazu, lange Ausdrücke lesbarer zu machen. Deshalb bietet es sich an, auch bei Typausdrücken eine Möglichkeit vorzusehen, die Charakterisierung der beteiligten Typvariablen aus dem eigentlichen Ausdruck herauszuziehen. Das folgende Beispiele illustriert drei gleichwertige Schreibweisen eines polymorphen Typs:

$$\begin{array}{ll} \mathit{length}{:}\ \alpha{:}\,\mathit{Type} \mapsto \mathit{List}\ \alpha \to \mathit{Nat} & -\!-\ \textit{normale Notation} \\ \mathit{length}(\alpha{:}\,\mathit{Type}){:}\ \mathit{List}\ \alpha \to \mathit{Nat} & -\!-\ \textit{musterbasierte Notation} \\ \mathit{length}{:}\ \mathit{List}\ \alpha \to \mathit{Nat}\quad \text{VAR}\ \alpha{:}\,\mathit{Type} & -\!-\ \textit{nachgestellte Notation} \end{array}$$

Die letzte Notation entspricht dem ... WHERE $x = \ldots$ bei normalen Ausdrücken. Aber weil wir jetzt keine Definitionsgleichungen haben, sondern nur Typannotationen, verwenden wir ein anderes Schlüsselwort.

Definition (nachgestellte Typvariablen)

Die beiden folgenden Typisierungen sind gleichwertig:

FUN $f{:}\ T_x$ VAR $x{:}\,\mathit{Type}$ ist gleichwertig zu FUN $f{:}\ x{:}\,\mathit{Type} \mapsto T_x$

Das Entsprechende gilt für alle Arten von Variablen in abhängigen Typen.

8.3.3 *Syntactic sugar*: Optionale Parameter

Wir haben abhängige Typen auf homogene Weise in unsere funktionale Welt eingebaut, indem wir sie als Funktionen repräsentieren, die Typen oder Werte in Typen abbilden. Allerdings müssen wir noch ein weiteres pragmatisches Problem lösen, das im folgenden Beispiel illustriert wird:

FUN $\mathit{length}{:}\ \alpha{:}\,\mathit{Type} \mapsto \mathit{List}\ \alpha \to \mathit{Nat}$
DEF $\mathit{length}(\alpha)(s) =$ IF s IS Empty THEN 0
ELSE $1 + \mathit{length}(\alpha)(\mathit{rt}\ s)$ FI

Wenn wir z. B. eine Liste von reellen Zahlen haben, also $s{:}\,\mathit{List}\ \mathit{Real}$, dann müssen wir die Anwendung der *length*-Funktion folgendermaßen schreiben:

$\ldots\,\mathit{length}(\mathit{Real})(s)\ldots$

Der Zwang, immer den Typ hinzufügen zu müssen, macht die Notation relativ hässlich, wie man an der Definition von *length* erkennen kann. Doch diese

Angabe ist fast immer redundant, weil sich der Typ aus dem des Arguments s ablesen lässt. Daher treffen wir folgende *notationelle Verabredung.*

Definition (optionale Typparameter)

Wenn die Typparameter in eckige Klammern eingeschlossen werden, also z. B.

FUN $length\colon [\alpha\colon \mathit{Type}] \mapsto \mathit{List}\ \alpha \to \mathit{Nat}$

oder

FUN $lenght[\alpha\colon \mathit{Type}]\colon\ \mathit{List}\ \alpha \to \mathit{Nat}$

dann dürfen die Argumente bei der Applikation weggelassen werden, also z. B.

$\ldots\, \mathit{length}(s) \ldots$ ist äquivalent zu $\ldots\, \mathit{length}[\mathit{Real}](s) \ldots$

Die Möglichkeit, den Typparameter bei Bedarf auch angeben zu können, ist eine pragmatische Notwendigkeit. Denn die Erfahrung zeigt, dass der Compiler manchmal überfordert ist, wenn er den einschlägigen Typ deduzieren soll, oder – noch schlimmer – dass die Applikation tatsächlich eine mehrdeutige Typisierung hat. In solchen Fällen muss der Programmierer die Chance haben, durch entsprechende explizite Typannotationen den Konflikt auflösen zu können.

Die optionalen Parameter sind natürlich nicht auf Typparameter beschränkt, sondern betreffen alle Arten von abhängigen Typen. Als Beispiel ändern wir die eingangs vorgeführte Funktion zur Berechnung des Skalarprodukts entsprechend ab:

FUN $\mathit{skalProd}[n\colon \mathit{Nat}]\colon\ \mathit{Vector}[n] \times \mathit{Vector}[n] \to \mathit{Real}$
TYPE $\mathit{Vector}[n\colon \mathit{Nat}] = (\mathit{Array}\ \mathit{Real}\ \mid\ \lambda\, a \bullet a.\mathit{length} = n)$

Dann darf man bei der Applikation der Funktion *skalProd* das Längenargument auch weglassen:

$\ldots\, \mathit{skalProd}(u, v) \ldots$ ist gleichwertig zu $\ldots\, \mathit{skalProd}[k](u, v) \ldots$

Voraussetzung ist natürlich, dass die beiden Vektoren u und v den gleichen Typ $\mathit{Vector}[k]$ haben.

Anmerkung: In der Sprache CAYENNE *gibt es die Idee der optionalen Typparameter ebenfalls. Diese werden dort durch den Pfeil* $\mapsto$ *gekennzeichnet (der in* CAYENNE *– anders als bei uns –* nicht *zur Annotation abhängiger Typen verwendet wird). Auch andere Sprachen, etwa* LEGO, QUEST *oder* RUSSELL *haben ähnliche Mechanismen entwickelt.*

9

Spezifikationen und Typklassen: Wie Typen typisiert werden

Es gibt keine Ideen, die nicht den
Stempel einer Klasse trügen.
Mao Zedong (Über die Praxis)

Wenn man ein gutes Argument hat, ergeben sich daraus Konsequenzen. Hat man zwei gute Argumente, ergeben sich mehr Konsequenzen als nur die Summe der beiden: man muss die Argumente in ihrem Zusammenspiel betrachten. Wir haben gesehen, dass die Idee der Typisierung ein gutes Mittel ist, um die korrekte Verwendung von Funktionen zu überprüfen. Wir haben auch gesehen, dass polymorphe Typen letztlich Funktionen sind, die Typen in Typen abbilden. Beide Beobachtungen zusammen verlangen nach einer „Typisierung von Typen“. In der Typtheorie werden solche Fragen auch untersucht; sie führen dort auf Begriffe wie „*higher-order polymorphism*“ und „*kinding*“ [117].

Auch in anderen Sprachen ist der Bedarf an dieser Art von Ausdrucksmöglichkeiten erkannt worden. So gibt es z. B. in JAVA das Konzept der *Interfaces*, die letztlich eine Typisierung von Klassen liefern.

In der Sprache HASKELL wurde ein ähnliches Konzept schon eingeführt: *Typklassen*. Allerdings war die Motivation eine ganz andere. Weil HASKELL keine ausgefeilten Mechanismen zur Einführung von überlagerten Mixfix-Operatoren hat, brauchte man einen geeigneten Ersatz. Dass dies nur eine partielle Lösung ist, sieht man z. B. daran, dass schon bald die Verallgemeinerung auf so genannte *Konstruktorklassen* eingeführt wurde, weil für bestimmte Anwendungen die Typklassen zu schwach waren.

Da dieser Stil – immer wenn mehr Mächtigkeit gebraucht wird, führt man eine neue Verallgemeinerung oder Variation ein – doch unbefriedigend ist, werden wir im Folgenden unserem durchgängigen Prinzip folgen und die Situation von vornherein ganz allgemein betrachten. Dabei zeigt sich schnell, dass Typklassen nur ein Spezialfall eines allgemeineren Konzepts sind, das aus der universellen Algebra kommt. Allerdings hat diese Allgemeinheit auch ihren Preis: Nicht alles, was man im Programm schreibt, kann der Compiler auch nachprüfen.

Algebraische Spezifikation

Zwei grundlegende Konzepte werden seit Jahrzehnten im Bereich der so genannten Algebraischen Spezifikationssprachen intensiv untersucht: *Signaturen* und *Spezifikationen*. (In [19, 101] wird der Versuch gemacht, die wesentlichen Aspekte dieses Themas im Rahmen eines umfassenden Sprachdesigns zu erfassen. Eine detaillierte Ausarbeitung des Themas aus stärker theoretischer Sicht findet sich z. B. in [43, 44].)

Aus der Typtheorie ist bekannt, dass sich diese algebraische Sichtweise auch in eine mehr typorientierte übertragen lässt; diese scheint sich auf den ersten Blick sogar harmonischer in die Welt der Funktionalen Programmierung einzufügen. Aber bei genauerem Hinsehen zeigen sich auch Nachteile: Wenn man den Einbau ganz systematisch und orthogonal zu den anderen Sprachkonzepten vornimmt, dann werden die Notationen ziemlich unleserlich. Als Konsequenz werden dann Ad-hoc-Notationen eingeführt, die schnell zu einer Art „Featuritis“ führen.

Demgegenüber hat der algebraische Ansatz den großen Vorteil, auf ganz natürliche Weise das softwaretechnische Prinzip der Modularisierung umzusetzen. Und diese Modularisierung macht die Notationen wesentlich lesbarer.

Aus diesem Grund werden wir versuchen, beide Welten miteinander zu verschmelzen. Das heißt, wir führen die algebraischen Konzepte der Signaturen und Spezifikationen explizit in die Sprache ein (was ganz leicht geht, weil beides nur weitere Inkarnationen von Gruppen sind), schlagen aber gleichzeitig die Brücke zur Typtheorie, indem wir Signaturen und Spezifikationen zur Typisierung von Strukturen verwenden. Als Nebenprodukt wird sich dabei auch klar zeigen, welche Rolle die Typklassen (die in HASKELL eingeführt wurden) spielen.

Motivation: Die Typisierung von Typen

Etwas zu tun, nur weil es sich als „natürliche Verallgemeinerung“ von etwas anderem ergibt, mag ein bewährter und erfolgreicher Arbeitsstil der Mathematik sein. In der Informatik reicht das als Begründung nicht aus. Hier muss man auch die Zusatzfrage stellen, ob es denn einen *Bedarf* für das verallgemeinerte Konzept gibt. Um das zu sehen, betrachte man wohlbekannte Funktionen wie das Sortieren von Sequenzen oder die Maximumsbildung von zwei Werten

FUN *sort*: $[\alpha\colon \mathit{Type}] \mapsto \mathit{Seq}\ \alpha \to \mathit{Seq}\ \alpha$ *−− reicht nicht!*
FUN *max*: $[\alpha\colon \mathit{Type}] \mapsto \alpha \times \alpha \to \alpha$ *−− reicht nicht!*

Solche Funktionen werden polymorph formuliert, genauer: mit abhängigen Typen, weil z. B. das Sortieren von Sequenzen von reellen Zahlen genauso programmiert wird wie das Sortieren von Sequenzen von Bankkunden. *Aber natürlich geht das so nicht.* Denn wir können nicht Sequenzen beliebiger Elemente sortieren. Irgendwo im Programmcode wird ein Vergleich von Sequenzelementen stattfinden müssen. Folglich lässt sich die Funktion *sort* nur für solche Komponententypen α programmieren, die einen passenden „<“-Operator

besitzen. Um diese Forderung präzise hinschreiben zu können, nehmen wir an, dass es eine „Typklasse“ *Ord* gibt, die genau diejenigen Typen beschreibt, die ein „<“ besitzen. Dann können wir schreiben

FUN *sort*: $[\alpha\colon \mathit{Ord}] \mapsto \mathit{Seq}\ \alpha \to \mathit{Seq}\ \alpha$
FUN *max*: $[\alpha\colon \mathit{Ord}] \mapsto \alpha \times \alpha \to \alpha$

Dabei verwenden wir für die Typklassen einen anderen Font, um die verschiedenen Ebenen der Typisierung besser auseinander halten zu können.

Subtypen oder Typklassen?

An dieser Stelle sieht sich der Sprachdesigner mit einer grundsätzlichen Entwurfsentscheidung konfrontiert. Viele Anwendungsprobleme lassen sich mit Hilfe von Subtypen ebenso lösen wie mit Hilfe von Typklassen.[1] Damit stellt sich die Frage, ob man das Konstrukt der Typklassen überhaupt einführen sollte. (Die Sprache CAYENNE zum Beispiel hat sich dagegen entschieden; sie löst alle Probleme mit *abhängigen Typen*.) Beides hat Vor- und Nachteile. In der ersten Variante braucht man nur ein einziges Konzept, nämlich Typen, aber dafür muss dieses Sprachmittel für ganz unterschiedliche Zwecke herhalten; diese Überfrachtung eines Konzepts ist manchmal eher verwirrend als hilfreich. In der zweiten Variante muss man sich mit mehreren Konzepten anfreunden; aber diese sind dann jeweils spezifischen Problemkreisen zugeordnet.

Da dieses Buch keine konkrete Sprache vorstellt, sondern sich mit grundlegenden Konzepten funktionaler Programmierung befasst, ergibt sich die Antwort von selbst: Typklassen sind ein fundamentales Konzept und müssen deshalb diskutiert werden.

Ein Stufenmodell der Typisierung

Damit ergibt sich die Situation, die in Abbildung 9.1 skizziert ist: Wir erhalten ein *gestuftes System der Typisierung* (horizontal), verbunden mit einem *gestuften System der Modularisierung* (vertikal).

- Der innerste Bereich betrifft die *Werte*. Dies sind atomare Werte wie Zahlen, Buchstaben etc., zusammengesetzte Werte wie Tupel, Listen, Arrays etc. und auch Funktionen.
- Die Welt der Werte wird durch *Typen* klassifiziert. Es gibt atomare Typen, Tupeltypen, Funktionstypen etc.
- Die Welt der Typen wiederum wird durch *Typklassen* klassifiziert. Dazu gehören Konzepte wie *Eq*, *Ord*, *Numeric* etc., die wir in diesem Kapitel genauer diskutieren werden.

[1] Man kann zu jeder Typklasse einen artifiziellen Typ assoziieren, der genau der Supertyp aller in der Klasse enthaltenen Typen ist.

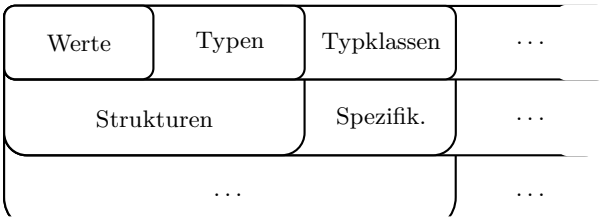


Abb. 9.1: Ein Stufensystem der Typisierung

- Auf der ersten Modularisierungsstufe erlauben *Strukturen* die Zusammenfassung von Werten und Typen zu neuen Einheiten.
- Diese Strukturen werden durch *Spezifikationen* typisiert.
- ... und so weiter.

Das Schichtenmodell aus Abbildung 9.1 lässt sich prinzipiell in beide Richtungen beliebig fortsetzen. Allerdings gibt es jenseits der hier aufgeführten Begriffe heute keine gebräuchlichen Instanzen, so dass es müßig wäre, weitere Namen zu erfinden.[2] Aus diesem Grund haben wir in Kapitel 4 den generischen Begriff der *Gruppen* eingeführt, mit dem sich die Modularisierung beliebig weiter fortsetzen lässt.

Werte	**Typen**	**Typklassen**
$2,\ (7.5, 12),\ \texttt{"}hallo\texttt{"},$ $sin,\ curry,\ \lambda x \bullet x + 1$	$Int,\ (Real, Real),$ $(Int \to Real),\ Seq(\_)$	$Eq,\ Ord,\ Numeric$ $Fun,\ Interval$

Wert gehört zum *Typ*	*Typ* gehört zur *Typklasse*
$2 : Int$	$Int : Ord$
$sin : (Real \to Real)$	$Set : Eq \to Type$

Tab. 9.1: Werte, Typen und Typklassen

Unsere Welt wird jetzt etwas reichhaltiger: (s. Tabelle 9.1). Die Elemente der verschiedenen Stufen sind durch die *Typisierungsrelation* miteinander verbunden. Zwischen Werten und Typen kennen wir das bereits sehr gut. Die entsprechende Relation zwischen Typen und Typklassen ist dagegen neu. Auf dieser Grundlage kann man sich auf eine sehr einfache, aber auch sehr allgemeine Sicht von Typklassen verständigen.

[2] In der Sprache CAYENNE [15] wird – angelehnt an Konzepte der Typtheorie [139] – unsere Klasse *Type* aller Typen als # geschrieben, genauer als $\#_1$. Diese Klasse wiederum ist von der Art $\#_2$, die selbst von der Art $\#_3$ ist. Und so weiter.

Definition (Typklasse, *Type*)

Eine ***Typklasse*** $\mathcal{C}$ charakterisiert eine Menge von Typen (genauso wie ein Typ eine Menge von Werten charakterisiert).

Die Klasse aller Typen bezeichnen wir mit *Type*.[3]

Hinweis zur Terminologie. Es gibt jetzt einen kleinen Konflikt zwischen Umgangssprache und technischen Begriffen. Auf der einen Seite haben wir das Sprachkonzept *Typ*, das z. B. in Abbildung 9.1 auf der zweiten Stufe der Hierarchie steht; wir nennen dies *Typ im engeren Sinn.* Auf der anderen Seite benutzen wir Formulierungen wie „Die Struktur A hat als Typ die Signatur S." In diesen Fällen sprechen wir von *Typ im weiteren Sinn.* Wir gehen im Folgenden davon aus, dass diese terminologische Überlagerung keine Probleme bereitet, weil im Kontext jeweils klar wird, welcher Typbegriff gemeint ist.

9.1 Operatoren auf Typklassen

Zweck und Idee der Typklassen sind ganz analog zu denen der Typen – nur eine Stufe höher. Deshalb ist es nicht überraschend, dass man zur Konstruktion von Typklassen dieselben Mechanismen hat wie bei Typen. Dazu gehören insbesondere (vgl. Kapitel 6)

- Aufzählung
- Produkt (Tupel und Gruppen)
- Summe
- Funktion (Generische Klassen)
- Rekursion

Wegen der Analogie zu Typen werden wir diese Mechanismen hier nicht nochmals diskutieren; dazu kommt, dass sie in der Praxis eine eher untergeordnete Rolle spielen.

Interessanter ist eine andere Frage: *Wie kommt man zu den elementaren Typklassen?* Denn die obigen Operatoren konstruieren – mit Ausnahme der Aufzählung – jeweils nur neue Typklassen aus bereits existierenden. Bei den Typen hat man als Basis traditionell die im Compiler vorgegebenen „Maschinen-Typen" wie *Char*, *Int*, *Real* etc. (vgl. Abschnitt 6.2). Einen vergleichbaren Ansatz gibt es auch bei Typklassen; so sind z. B. in HASKELL unter anderem die folgenden Klassen vordefiniert (die der Benutzer allerdings auch selbst definieren könnte):

- `Eq`: Typen mit Gleichheitsoperation
- `Ord`: Typen mit Ordnungsrelationen
- `Number`: Typen mit arithmetischen Operationen

[3] Die Typklasse *Type* ist vergleichbar mit der universellen Klasse `Class` in JAVA.

- `Show/Read`: Typen mit String-Repräsentationen

Das genügt aber sicher nicht für praktische Anforderungen; man muss dem Programmierer auch die Möglichkeit geben, eigene Typklassen einzuführen; die dazu notwendigen Mechanismen werden wir im Rest dieses Kapitels diskutieren. Wie wir sehen werden, lassen sich damit auch die oben erwähnten vorgegebenen Typklassen definieren, was sie weniger zufällig erscheinen lässt.

9.2 Signaturen: Die Typisierung von Strukturen

Signaturen sind ein zentrales Element der Algebraischen Spezifikationssprachen [19]. In die funktionale Programmierung wurden sie als explizites Sprachkonstrukt in ML und später auch in OPAL eingeführt. (Auch in JAVA gibt es dieses Konzept; es wird dort *Interface* genannt.) Und sie sind – wie wir gleich sehen werden – die Basis für die Typklassen von HASKELL. Signaturen dienen im Wesentlichen dazu, Strukturen zu typisieren.

Strukturen sind ein wichtiges Modularisierungsmittel, das zusammengehörige Typen und Funktionen zu einer Einheit zusammenfasst. (In der Mathematik werden sie *Algebren* genannt.) In unserem Kontext sind sie als spezielle Gruppen darstellbar. Zur Illustration der folgenden Konzepte verwenden wir das vertraute Beispiel der ganzen Zahlen, die in der Struktur *Integer* in Programm 9.1 beschrieben sind. Dabei drücken wir mit dem Schlüsselwort GIVEN aus, dass die Implementierung des Typs Int vom Compiler vorgegeben ist.

Programm 9.1 Die Struktur der ganzen Zahlen (Fragment)

```
STRUCTURE Integer = {
  ITEM Int: Type = GIVEN                          -- Implementierung ist vordefiniert
  ITEM _ + _: (Int × Int → Int) = ...             -- Addition
  ITEM _ · _: (Int × Int → Int) = ...             -- Multiplikation
       ...
  ITEM _ = _: (Int × Int → Bool) = ...            -- Gleichheitstest
  ITEM _ ≠ _: (Int × Int → Bool) = ...            -- Ungleichheitstest
  ITEM _ < _: (Int × Int → Bool) = ...            -- Kleinertest
       ...
}
```

Der Typ einer solchen Struktur wird als ihre *Signatur* bezeichnet (wobei wir das Wort „Typ“ jetzt in seiner allgemeineren Bedeutung verwenden). Für das Beispiel der ganzen Zahlen von Programm 9.1 bezeichnen wir die entsprechende Signatur als *IntegerSig*. Sie ist in Programm 9.2 angegeben. Um die Intuition zu unterstützen, verwenden wir die Schlüsselwörter TYPEOF bzw. KINDOF anstelle des nichtssagenden ITEM.

Programm 9.2 Die Signatur der ganzen Zahlen (Fragment)

```
SIGNATURE IntegerSig = {
  KINDOF Int = Type                          -- Typklasse von Int
  TYPEOF _ + _ = (Int × Int → Int)           -- Typ der Addition
  TYPEOF _ · _ = (Int × Int → Int)           -- Typ der Multiplikation
         ...
  TYPEOF _ = _ = (Int × Int → Bool)          -- Typ des Gleichheitstests
  TYPEOF _ ≠ _ = (Int × Int → Bool)          -- Typ des Ungleichheitstests
  TYPEOF _ < _ = (Int × Int → Bool)          -- Typ des Kleinertests
         ...
}
```

Wie man an diesem Beispiel sofort erkennen kann, erfasst die Signatur gerade den Typisierungsanteil der Struktur. Das heißt, die Signatur ist ebenfalls eine Gruppe und sie hat die gleichen Selektoren wie die Struktur. Die selektierten Elemente sind jetzt allerdings nicht die Werte, sondern die zugehörigen Typen.

Wir verwenden solche Signaturen, um die entsprechenden Strukturen zu typisieren. Im obigen Beispiel der Programme 9.1 und 9.2 sieht diese Typisierungsrelation folgendermaßen aus:

Integer : *IntegerSig* -- *Typisierung der Struktur Integer*

Es gibt aber eine Komplikation! Wenn wir die Definition von Gruppen betrachten, dann erhalten wir ein Bindungsproblem: In Programm 9.2 bezieht sich nach den üblichen Bindungsregeln der Name *Int* in den Funktionalitäten der Operationen auf *IntegerSig*.*Int* – und das ist *Type*. Gemeint ist aber ein konkreter Typ (und zwar der, der in *Integer*.*Int* definiert wird). Diese Beobachtung zeigt, dass Signaturen nicht naiv als Gruppen aufgefasst werden können, sondern eine etwas ausgefeiltere Erklärung brauchen. Wir stellen diesen Punkt aber zurück bis Abschnitt 9.4, weil er auch das Konzept der Spezifikationen betrifft, die wir vorher noch einführen wollen.

9.2.1 *Syntactic sugar*: Traditionelle Signatur-Notation

Wir hatten bei der Einführung der Gruppen in Kapitel 4 als einheitliche Syntax festgelegt, dass die Zuordnung zwischen den Selektoren und den Werten, für die sie stehen, mit einem Gleichheitszeichen geschrieben wird. Dieser Konvention genügen auch die Struktur und die Signatur in den obigen Programmen 9.1 und 9.2.

Aber Signaturen werden traditionell meistens anders geschrieben: Die Operationssymbole werden ihren Funktionalitäten mit einem Doppelpunkt zugeordnet. Um hier den Konflikt zwischen konzeptuell richtiger und traditionell üblicher Notation aufzulösen, können wir spezielle Schlüsselwörter verwenden.

Festlegung (Die Schlüsselwörter FUN und TYPE)

In einer Signatur sind die folgenden Notationen gleichwertig:

FUN «*name*»: «*Typ*»	entspricht	TYPEOF «*name*» = «*Typ*»
	bzw.	ITEM «*name*» = «*Typ*»
TYPE «*name*»	entspricht	KINDOF «*name*» = *Type*
	bzw.	ITEM «*name*» = *Type*

Die Schlüsselwörter TYPEOF und KINDOF sind nur mnemotechnisch aussagekräftigere Synonyme für ITEM.

Mit dieser notationellen Vereinbarung lässt sich die Signatur aus Programm 9.2 in eher traditioneller Form schreiben:

```
SIGNATURE IntegerSig = {
  TYPE Int
  FUN _ + _: Int × Int → Int      -- Typ der Addition
  FUN _ · _: Int × Int → Int      -- Typ der Multiplikation
      ...
  FUN _ = _: Int × Int → Bool     -- Typ des Gleichheitstests
  FUN _ ≠ _: Int × Int → Bool     -- ...
  FUN _ < _: Int × Int → Bool
      ...
}
```

Wir werden im Folgenden die konzeptuelle und die traditionelle Notation gleichrangig nebeneinander benutzen.

9.2.2 *Syntactic sugar*: Verschmelzen von Struktur und Signatur

Die Aufspaltung von Strukturen und ihren Signaturen in zwei getrennte Gruppen ist aus softwaretechnischen Gründen nicht immer empfehlenswert. Man sieht das Problem schon deutlich an den kleinen Beispielen *Integer* und *IntegerSig* in Programm 9.1 und 9.2. (Diese Beobachtung wird auch durch die schlechte Erfahrung mit der Sprache OPAL gestützt, die eine solche Trennung zwingend fordert.) Aus Dokumentations- und Lesbarkeitsgründen empfiehlt es sich, die Typinformation auch in der Struktur explizit zu machen.

Deshalb sollte man grundsätzlich erlauben, die Struktur und die Signatur miteinander zu verschmelzen, so dass man nicht gezwungen ist, die Typinformation immer an den Definitionen zu annotieren. Im Beispiel *Integer* sieht das Ergebnis einer solchen Verschmelzung folgendermaßen aus (unter Verwendung der traditionellen Notation):

```
STRUCTURE Integer = {
  TYPE Int
  DEF Int = GIVEN
  FUN _ + _: Int × Int → Int      -- Addition
  DEF a + b = ...
  FUN _ · _: Int × Int → Int      -- Multiplikation
  DEF a · b = ...
      ...
  FUN _ = _: Int × Int → Bool     -- Gleichheitstest
  DEF (a = b) = ...
  FUN _ ≠ _: Int × Int → Bool     -- ...
  DEF (a ≠ b) = ...
  FUN _ < _: Int × Int → Bool
  DEF (a < b) = ...
      ...
}
```

Wie man an diesen wenigen Beispielen schon erkennen kann, gibt es auf der Ebene der Modularisierungskonzepte eine Fülle von Notationen. Aber sie lassen sich alle als *Syntactic sugar* verstehen, hinter dem immer dasselbe Konzept steht: *Signaturen sind die Typisierung von Strukturen.* (Diese grundsätzliche Sichtweise kann man schon in ML finden und sie wird z. B. in [118] aus typtheoretischer Sicht genauer beleuchtet.)

9.3 Spezifikationen: Subtypen von Signaturen

Im Bereich der Algebraischen Spezifikation werden Strukturen (also Algebren) axiomatisch über ihre Eigenschaften beschrieben. Im Kontext von Typisierungskonzepten entspricht dieses Vorgehen der Einführung von Subtypen. Wir könnten z. B. folgende Spezifikation schreiben:

```
SPECIFICATION IntegerSpec = ( IntegerSig | «Properties» )
```

In den «*Properties*» würden dann Eigenschaften wie Kommutativität, Assoziativität etc. aufgelistet. Allerdings würde sich bei einer konkreten Umsetzung dieser Idee schnell zeigen, dass die Notation sehr unleserlich wird. (Man landet sofort bei notationellen Ungetümen, die bei den Instanziierungen von Interfaces in JAVA berüchtigt sind.) Aus diesem Grund wählt man besser lesbare Notationen, wie sie im Programm 9.3 exemplarisch gezeigt sind.

Anstelle der Erweiterung mittels EXTEND *IntegerSig* hätte man in Programm 9.3 die Signatur auch explizit hinschreiben können. Aber in dieser modularisierten Form ist die Beschreibung softwaretechnisch besser.

Wir werden häufig auf die Angabe des expliziten Allquantors in den Propertys verzichten; das stellt zwar hohe Anforderungen an den Compiler, der dann die Variablen in der Formel selbst erkennen muss, aber es erhöht im Allgemeinen die Lesbarkeit für menschliche Nutzer.

Programm 9.3 Die Spezifikation der ganzen Zahlen (Fragment)

```
SPECIFICATION IntegerSpec = {
  EXTEND IntegerSig
  PROP ∀a, b • a + b = b + a                    -- Kommutativität
  PROP ∀a, b, c • (a + b) + c = a + (b + c)     -- Assoziativität
       ...
  PROP ∀a, b, c • (a + b) · c = (a · c + b · c)  -- Distributivität
       ...
}
```

In manchen Sprachen lassen sich die Propertys auch benennen, so dass man sich z. B. in Beweisen oder in der Dokumentation besser auf sie beziehen kann. (Wir werden in Kapitel 16 ausgiebig von dieser Möglichkeit Gebrauch machen.) Ein typisches Beispiel wäre etwa die Assoziativität in Programm 9.3:

PROP *Assoc* IS $\forall a, b, c \bullet (a + b) + c = a + (b + c)$ -- *Assoziativität*

Mit der Spezifikation in Programm 9.3 lässt sich die Typisierung der Struktur *Integer* jetzt präziser fassen als nur mit der Signatur:

Integer: *IntegerSpec* -- *genauere Typisierung der Struktur Integer*

Definition (Spezifikation)

Eine ***Spezifikation*** ist eine Signatur, die um *Propertys* angereichert ist. Dies entspricht einer – notationell eleganter dargestellten – Subtyp-Konstruktion. (Signaturen sind damit Spezialfälle von Spezifikationen.)

Spezifikationen dienen (wie Signaturen) dazu, Strukturen zu typisieren.

Die Welt der Spezifikationen ist so umfangreich, dass man ganze Bücher mit ihnen füllen kann (was auch getan wurde). Wir beschränken uns im weiteren Verlauf des Kapitels auf ein paar wichtige und typische Beispiele. Diese Beispiele machen aber – trotz ihrer geringen Anzahl und Größe – einen ganz wichtigen Aspekt deutlich. Auch (oder gerade) auf der Ebene der Signaturen und Spezifikationen ist eine gut durchdachte *Modularisierung* zwingend erforderlich. Denn nur so lassen sich Konzepte, Ideen und Lösungen tatsächlich wiederverwenden.[4]

Anmerkung: Die Sprache SPECWARE *und das zugehörige Entwicklungssystem [6, 8], die am Kestrel Institute entwickelt wurden, zeigen, dass man in dieser Richtung sehr weit gehen kann. Dort werden Softwaresysteme als eine Vielzahl von (kleinen) Spezifikationen entwickelt, die über vielfältige Morphismen und ähnliche Konzepte miteinander verwoben sind und auseinander hervorgehen.*

[4] Diese Forderung wird bekanntlich im Software-Engineering gebetsmühlenartig wiederholt, aber – mangels geeigneter Sprachmittel – faktisch so gut wie nie umgesetzt.

Spezifikationen sind nicht testbar! Es gibt einen fundamentalen Unterschied zwischen den echten Subtypen aus Abschnitt 7.2 und den Spezifikationen: Bei einem Subtyp $(T \mid p)$ ist das Constraint p im Allgemeinen zumindest zur Laufzeit prüfbar. Spezifikationen dagegen basieren im Allgemeinen auf Propertys, die höchstens beweisbar sind, aber nicht testbar. Damit führen sie streng genommen aus dem Rahmen der funktionalen Programmierung hinaus.

9.4 Signaturen und Spezifikationen sind existenziell

Wir hatten schon in Abschnitt 9.2 darauf hingewiesen, dass Signaturen und Spezifikationen nicht in naiver Weise als Gruppen aufgefasst werden können, weil dies zu falschen Bindungen führen würde. Diese Problematik wollen wir jetzt genauer studieren. Zur Illustration verallgemeinern wir dazu das Beispiel der Zahlen aus den Programmen 9.1, 9.2 und 9.3.

9.4.1 Zahlen generell betrachtet

Was wir bei den ganzen Zahlen $\mathbb{Z}$ gemacht haben, funktioniert genauso bei den natürlichen Zahlen $\mathbb{N}$, den reellen Zahlen $\mathbb{R}$ oder den komplexen Zahlen $\mathbb{C}$. Allerdings wäre es sehr unökonomisch, wenn wir jeweils die Spezifikationen duplizieren würden, wobei immer nur der Typname ausgetauscht würde: einmal *Nat*, einmal *Int*, einmal *Real* und einmal *Complex*. Ein solches Vorge-

Programm 9.4 Die Spezifikation von Zahlen (Fragment)

```
SPECIFICATION Number = {
  TYPE Num
  FUN _ + _: Num × Num → Num      -- Addition
  FUN _ · _: Num × Num → Num      -- ...
      ...
  FUN _ = _: Num × Num → Bool
  FUN _ ≠ _: Num × Num → Bool
  FUN _ < _: Num × Num → Bool
      ...
  PROP a + b = b + a              -- Kommutativität
  PROP (a + b) + c = a + (b + c)  -- Assoziativität
       ...
}
```

hen widersprüche allen Regeln modernen Software-Engineerings. Stattdessen sollten wir die Spezifikation nur einmal aufschreiben und dabei den Typ „generisch“ lassen. Dies ist in Programm 9.4 illustriert.

Jetzt können wir die entsprechenden Strukturen typisieren, zum Beispiel:[5]

STRUCTURE *Natural* : (*Number* RENAMING *Num* AS *Nat*) = {...}
STRUCTURE *Integer* : (*Number* RENAMING *Num* AS *Int*) = {...}
STRUCTURE *Real* : (*Number* RENAMING *Num* AS *Real*) = {...}
STRUCTURE *Complex* : (*Number* RENAMING *Num* AS *Complex*) = {...}

Die Vorteile dieses Vorgehens sind evident. Man muss die vielen Funktionalitäten und Gesetze wie Kommutativität, Assoziativität etc. nicht immer wieder hinschreiben, sondern hat sie ein für alle Mal fixiert. Das macht nicht nur das Arbeiten mit den Strukturen einfacher und kompakter, sondern trägt auch ganz wesentlich zur Erhöhung des Dokumentationswertes bei.

Vor allem kann man ein zentrales Qualitätskriterium moderner Software erfüllen: Programme lassen sich auf einem adäquaten Abstraktionsniveau formulieren. Denn viele Algorithmen funktionieren für alle möglichen Arten von Zahlen. Ein typisches Beispiel ist das Skalarprodukt von Vektoren:

FUN $skalProd: [n: Nat] \mapsto Vector[n][Num] \times Vector[n][Num] \to Num$
DEF $skalProd(u, v) = +\,/\,(u \dot{\bigvee} v)$

Dieses Programm funktioniert bei allen Arten von Vektoren, sei es über ganzen, reellen oder komplexen Zahlen.

9.4.2 Existenzielle Typen

Wir hatten bereits erwähnt, dass es mit Namen wie *Num* in Programm 9.4 ein Bindungsproblem gibt, wenn wir eine naive Gruppensicht zugrunde legen. Dahinter steckt aber mehr als nur ein oberflächliches syntaktisches Scopingproblem, wie man aus der Typtheorie [117] lernen kann.

Es gibt einen fundamentalen Unterschied zwischen dem Namen *Int* in der konkreten Struktur *Integer* und dem Namen *Num* in der abstrakten Spezifikation *Number*: Der Name *Int* steht für einen ganz konkreten Typ, der Name *Num* dagegen für viele mögliche Typen – unter anderem *Nat*, *Int*, *Real*, *Complex* usw. Deshalb wird *Num* auch als ***abstrakter Typ*** bezeichnet.

Diese Beobachtung führt in der Typtheorie dazu, dass abstrakte Typen wie *Num existenziell quantifiziert* werden. In einer Ad-hoc-Notation könnten wir dies folgendermaßen schreiben:

```
SPECIFICATION Number =
  ∃Num: Type • {                          -- Ad-hoc-Notation!
    FUN _ + _: Num × Num → Num
    FUN _ · _: Num × Num → Num
    ...
  }
```

[5] In HASKELL gibt es eine ähnliche Strukturierung der zahlartigen Datentypen, allerdings mit Hilfe von so genannten Typklassen (s. Abschnitt 9.5.2). Diese Strukturierung ist wesentlich filigraner als unsere kleine Skizze.

Die konkreten Typen *Nat*, *Int*, *Real*, *Complex* etc. sind dann so genannte *Witnesses*[6] für die existenzielle Typvariable *Num*. Diese Interpretation müssen wir einer Typisierungsaussage der Form

Integer : (*Number* RENAMING *Num* AS *Int*)

zugrunde legen. Sobald wir dies machen, stimmen auch alle Bindungen in den Funktionalitäten, weil sich jetzt *Int* tatsächlich auf *Integer*.*Int* bezieht.

Auch wenn die semantische Fundierung auf existenziellen Typen basiert, wollen wir doch an den traditionellen Notationen wie z. B. in Programm 9.4 festhalten.

Festlegung (Schreibweise für Signaturen)

Wir schreiben Signaturen und Spezifikationen in der traditionellen Form, bei der die abstrakten Typen innerhalb der Gruppe aufgeführt werden. Semantisch entspricht das aber einer existenziellen Quantifizierung.

9.5 Von Spezifikationen zu Typklassen

Bisher haben wir uns primär mit der *Deklaration* von Signaturen und Spezifikationen befasst und die Frage ihrer *Verwendung* zurückgestellt. Typische Beispiele wären etwa die am Anfang des Kapitels erwähnte Funktion *sort*, die Sequenzen von beliebigen Elementen sortiert, vorausgesetzt auf dem Elementtyp existiert eine Ordnung, oder die Funktion *skalProd*, die für Vektoren mit beliebigen numerischen Typen funktioniert.

Mit unseren bisherigen Sprachmitteln ist das nicht gut ausdrückbar. Deshalb brauchen wir ein weiteres Konzept, nämlich die Typklassen im Sinne von HASKELL. Allerdings werden wir sie nicht als unabhängiges Feature einführen, sondern abgeleitet aus dem Begriff der Signaturen und Spezifikationen.

9.5.1 Typklassen à la HASKELL

Als Einstieg rekapitulieren wir kurz die Art und Weise, wie Typklassen in HASKELL eingeführt werden. Dazu orientieren wir uns an den Beispielen der wohl bekannten HASKELL-Klassen *Eq* und *Ord*. (Man beachte, dass in HASKELL Typisierung mit „ :: “ geschrieben wird.)

class Eq a where
$(==) :: a \to a \to Bool$
$(\neq) :: a \to a \to Bool$
$x \neq y = not\ (x == y)$

class Eq a $\Rightarrow$ *Ord a where*
$(<) :: a \to a \to Bool$
$(>) :: a \to a \to Bool$
...

[6] Wir ziehen den englischen Fachterminus der Übersetzung „Zeuge“ vor.

Für die Operation $\neq$ wird eine Default-Implementierung mit angegeben. Auf der rechten Seite wird außerdem noch das Prinzip der Vererbung demonstriert. Die Klasse *Ord* erweitert die Klasse *Eq* um eine Reihe weiterer Operationen; aber sie enthält auch die beiden Operationen von *Eq*.

Damit bleibt noch das Problem zu klären, wie ein Typ zum Mitglied einer Klasse wird. Das geschieht in HASKELL mit der *instance*-Anweisung.

$$\begin{aligned} &\mathit{instance}\ \mathit{Eq}\ \mathit{Nat}\ \mathit{where} \\ &\qquad x == y = \mathit{natEq}\ x\ y \end{aligned}$$

Schließlich werden diese Typklassen benutzt, um polymorphe Funktionen adäquat einzuschränken. (In HASKELL steht $[a]$ für unser $\mathit{List}\ \alpha$.)

$$\mathit{sort} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [a]$$

Wie dieses Beispiel zeigt, geht es bei den HASKELL-Typklassen primär um das Bereitstellen von Operationssymbolen, also – in unserem Sinne – um *Signaturen*. Damit unterscheidet sich der Ansatz von HASKELL von unseren bisherigen Konzepten in zwei Punkten:

- Der Begriff der Signatur wird nicht explizit verfügbar gemacht. Insbesondere wird keine Beziehung zwischen Signaturen und Moduln hergestellt.
- Dafür wird etwas anderes explizit bereit gestellt, was bei uns noch fehlt, nämlich ein Begriff für die Menge aller Typen, die in den Strukturen einer Signatur (bzw. Spezifikation) definiert werden.

9.5.2 Definition von Typklassen

Wie man an dem Vergleich zu HASKELL sehen kann, hängt die Idee der Typklassen ganz eng mit dem Konzept der Signaturen zusammen; damit lässt sich unsere erste Definition vom Anfang des Kapitels technisch präzisieren:

Definition (Typklasse)

Sei eine Signatur bzw. eine Spezifikation S gegeben, in der unter anderem ein abstrakter (also existenzieller) Typ α definiert wird. Dann ist die ***von*** S ***induzierte Typklasse*** die Menge aller Witness-Typen für α.

Damit bleibt die Frage, wie diese semantische Definition in eine geeignete Syntax gegossen werden kann. Programm 9.5 illustriert die Antwort anhand unseres laufenden Beispiels der Zahlen.

Aufgrund der obigen semantischen Überlegungen ist klar, dass die Typklasse *Numeric* nicht innerhalb der Spezifikation *Number* definiert werden kann, sondern gleichberechtigt neben ihr stehen muss. Die „Definitionsgleichung“ $\mathit{Numeric} = \mathit{Number}.\mathit{Num}$ ist zwar syntaktisch etwas gewagt, drückt aber doch ganz gut die semantische Eigenschaft aus, dass *Numeric* für alle konkreten Witnesstypen der existenziellen Typvariablen *Num* steht.

Programm 9.5 Die Typklasse *Numeric*

```
TYPECLASS Numeric = Number.Num
SPECIFICATION Number = {
  TYPE Num: Numeric
  FUN _ + _: Num × Num → Num        -- Addition
  FUN _ · _: Num × Num → Num        -- ...
  ...
  PROP a + b = b + a                -- Kommutativität
  PROP (a + b) + c = a + (b + c)    -- Assoziativität
  ...
}
```

Innerhalb der Spezifikation *Number* können wir jetzt den abstrakten Typ *Num* sogar mit seiner Typklasse „typisieren“. Man beachte jedoch, dass wir in Ausdrücken der Art *Number.Num* oder (*Number* RENAMING *Num* AS *Int*) ein bisschen mogeln, indem wir den Namen *Num* wie einen gewöhnlichen Gruppenselektor benutzen. Dabei ignorieren wir die semantische Eigenschaft, dass *Num* eine existenziell quantifizierte Variable ist. (Für einen Compiler ist es aber kein grundsätzliches Problem, mit dieser Dichotomie umzugehen.)

Diese Notation hat den kleinen Nachteil, dass *Number* und *Numeric* nicht zwingend beieinander stehen müssen, obwohl sie wechselseitig voneinander abhängen. Aber dafür gibt es eine Reihe von Vorteilen:

- Die Notation spiegelt die tatsächlichen semantischen Gegebenheiten wider.
- Man kann Signaturen einführen, ohne ihnen Typklassen zuzuordnen.
- Man kann Signaturen einführen, die mehrere Typen beschreiben und deshalb auch mehrere Typklassen induzieren. (Ein bekanntes Beispiel sind Bäume: Ein Baum ist ein Knoten mit einem Wald von Unterbäumen; und ein Wald ist eine Sequenz von Bäumen.) Ähnliche Konzepte werden in HASKELL unter dem Stichwort „Multi-Parameter-Typklassen“ diskutiert.

Es ist instruktiv, nochmals die Notation von HASKELL mit der unseren zu vergleichen. Tabelle 9.2 zeigt die wesentlichen Elemente anhand des Beispiels der rationalen Zahlen. Um die Entsprechung möglichst deutlich zu machen, lassen wir die Signatur anonym, indem wir den Selektor *Num* direkt auf die entsprechende Gruppe anwenden (was allerdings unter softwaretechnischen Strukturierungsgründen schlechter Programmierstil ist).

Man sieht, dass – bis auf ein paar Schlüsselwörter – beide Ansätze von der Aufschreibung her nahezu identisch sind. Der wesentliche Unterschied ist notationell kaum sichtbar, hat aber gravierende konzeptuelle Auswirkungen: Bei uns sind Typklassen ein abgeleitetes Konstrukt der Signaturen und Spezifikationen. Diese wiederum sind zentrale Mittel zur Modularisierung von Software. Und sie sind theoretisch gut in der Algebraischen Spezifikation untersucht.

Definition	
TYPECLASS *Numeric* = *Number*.*Num* SPECIFICATION *Number* = { TYPE *Num*: *Numeric* FUN _ + _: *Num* × *Num* → *Num* ... }	*class Numeric num* *where* (+) :: *num* → *num* → *num* ...
Instanziierung	
STRUCTURE *Rational*: *Number* = { TYPE *Rat* = (*Int* × *Nat*) DEF *a* + *b* = ... }	*data Rat* = *Fraction Int Nat* *instance Numeric Rat where* *a* + *b* = ...
Buch	HASKELL

Tab. 9.2: Typklassen im Vergleich mit HASKELL

Damit wird es leichter, die Idee der Typklassen auch auf andere Aspekte zu verallgemeinern. Ein typisches Beispiel ist die „Vererbung“ von Typklassen. Hier brauchen wir kein neues Konzept, weil die Standardmechanismen der Vererbung bei Gruppen und Subtypen auch auf Typklassen anwendbar sind. (Beispiele werden wir im Folgenden gleich sehen.)

Auch die so genannten *Konstruktorklassen* von HASKELL sind in unserem Ansatz bereits enthalten. Programm 9.6 enthält ein entsprechendes Beispiel. (In Kapitel 11 werden wir diese Beispiele genauer studieren.)

Programm 9.6 Die Typklasse *Functor*

```
TYPECLASS Functor = Functor.F
SPECIFICATION Functor = {
  TYPE F: Type → Type
  FUN _*: α: Type × β: Type ↦ (α → β) → (F α → F β)   -- Map
  ...
}
```

Anmerkung: Auch unser Ansatz hat Grenzen. Ein interessantes Beispiel ist das so genannte Polytypic programming [87, 16]. Diesem Konzept liegt die Idee zugrunde, dass z. B. die Instanziierung der Typklasse Functor für Listen, Mengen, Bäume, Arrays usw. immer dem gleichen Muster folgt: die Rekursionsstruktur der Typdeklaration spiegelt sich in der Rekursionsstruktur der Map-Funktion wider. Da hier die syntaktische Struktur der Programmterme essenziell ist, lässt sich das höchstens mit Reflection-Techniken realisieren.

9.6 Beispiele für Spezifikationen und ihre Typklassen

Es gibt eine Reihe von wichtigen Typklassen, die bei der Programmierung immer wieder gebraucht werden. Als typisches Beispiel haben wir bereits die Typklasse *Ord* angesprochen, die z. B. beim Sortieren benötigt wird. Im Folgenden wollen wir einige dieser Typklassen und ihre zugehörigen Spezifikationen exemplarisch vorführen.

9.6.1 Gleichheit

Auf praktisch allen Typen stehen Gleichheitstests zur Verfügung.[7] (Prominenteste Ausnahme von dieser Regel sind die Funktionstypen.) Aufgrund dieser universellen Verfügbarkeit lohnt es sich, die Gleichheit – und die mit ihr untrennbar verbundene Ungleichheit – in eine Spezifikation zu fassen. In Programm 9.7 geben wir die Typklasse und die Spezifikation an.

Programm 9.7 Die Spezifikation des Gleichheitstests

```
TYPECLASS Eq = Equality.α
SPECIFICATION Equality = {
  TYPE α: Eq
  FUN _ = _: α × α → Bool                  -- Gleichheitstest
  FUN _ ≠ _: α × α → Bool                  -- Ungleichheitstest
  PROP (a = a)                             -- reflexiv
  PROP (a = b) = (b = a)                   -- kommutativ
  PROP (a = b) ∧ (b = c) ⟹ (a = c)         -- transitiv
  PROP (a = b) ⟹ f(a) = f(b)               -- kongruent
  DEF (a ≠ b) = ¬(a = b)                   -- Default-Definition
}
```

Wir verwenden hier einige Konventionen, die kurz erläutert werden müssen:

- Das Gleichheitszeichen ist hier überlagert. Zum einen ist es die „Meta-Gleichheit“ der Sprache selbst, d. h. dasjenige Gleichheitszeichen, das wir in Definitionen, Axiomen etc. verwenden. Zum anderen ist es der in der Signatur eingeführte Testoperator, also eine von uns definierte Funktion. In vielen Sprachen werden die beiden Gleichheiten unterschieden, indem man die eine als „ = “ schreibt und die andere als „ == “ (wobei die jeweilige Zuordnung in den verschiedenen Sprachen unterschiedlich ausfällt). Wir denken, dass sowohl die Leser als auch die Compiler clever genug sein sollten, beides trotz der Überlagerung auseinander halten zu können.

[7] In der Sprache ML hat man deshalb als spezielle Sprachfeatures *eqtypes* und *restricted polymorphism* eingeführt.

- Wir gehen auch sehr freizügig mit der Beziehung zwischen Werten vom Typ *Bool* und der (Meta-)Wahrheit von Formeln um. Streng genommen müssten wir an ein paar Stellen noch ein „$\ldots = \mathit{true}$“ anfügen.
- Schließlich verwenden wir noch eine Idee von HASKELL: Wenn in einer Spezifikation anstelle von PROP ein DEF geschrieben wird, liefert das eine *Default-Definition*. Das heißt, wenn in einer Struktur dieser Spezifikation keine eigene Definition angegeben ist, wird die Default-Definition aus der Spezifikation genommen.

Wie schon in Kapitel 7 ausgiebig diskutiert, erlaubt die Subtyp-Relation, dass Subgruppen „zu viele“ Komponenten haben. Deshalb gelten z. B. die folgenden Typisierungen für Strukturen wie *Integer* oder *Real*.

$\mathit{Integer}$: $\mathit{Equality}$ RENAMING α AS Int
Real : $\mathit{Equality}$ RENAMING α AS Real
...

In der Praxis gilt diese Typisierung mittels *Equality* für nahezu alle Strukturen. Deshalb hat z. B. HASKELL das Schlüsselwort *deriving* eingeführt, mit dem man derartige Standardfälle automatisch vom Compiler erledigen lassen kann. (Allerdings wird dieser Mechanismus in HASKELL nicht für Strukturen oder Moduln benutzt, sondern für Typklassen.)

9.6.2 Ordnung

Nach der Gleichheit dürfte die zweithäufigste Eigenschaft von Daten die Ordnung sein. Dabei kann man meistens sogar eine lineare Ordnung erwarten. Trotzdem bauen wir unsere Beispiele etwas filigraner auf, indem wir mit partiellen Ordnungen beginnen.

Das Programm 9.8 zeigt bereits einige Möglichkeiten, Spezifikationen in modularisierter Form aufzuschreiben:

- Beide Spezifikationen *PartialOrder* und *Order* haben die gleiche Signatur. Man könnte allerdings bei *Order* den Komfort erhöhen und noch die Operationen $\mathit{min}(a, b)$ und $\mathit{max}(a, b)$ hinzunehmen.
- Beide Spezifikationen schließen die Gleichheit *Equality* mit ein (wodurch auch die (existenzielle) Typvariable α eingeführt wird.
- Die Spezifikation *Order* erweitert nur die Menge der Eigenschaften von *PartialOrder*, fügt aber keine neuen Operationen oder Typen hinzu. (Deshalb haben beide die gleiche Signatur.)

Wie schon bei *Equality* gilt auch hier für die meisten Strukturen, dass sie vom Typ *Order* sind, z. B.:

$\mathit{Integer}$: Order RENAMING $(\alpha, \prec, \preceq, \ldots)$AS $(\mathit{Int}, <, \leq, \ldots)$
Real : Order RENAMING $(\alpha, \prec, \preceq, \ldots)$AS $(\mathit{Real}, <, \leq, \ldots)$
...

Programm 9.8 Die Spezifikation der partiellen und totalen Ordnung

```
SIGNATURE OrderSig = {
  EXTEND EqualitySig
  FUN _ ≺ _ : α × α → Bool
  FUN _ ⪯ _ : α × α → Bool
  FUN _ ≻ _ : α × α → Bool
  FUN _ ⪰ _ : α × α → Bool
}
TYPECLASS POrd = PartialOrder.α
SPECIFICATION PartialOrder = {
  EXTEND OrderSig, Equality
  PROP ¬(a ≺ a)
  PROP (a ≺ b) ∧ (b ≺ a) = false          −− asymmetrisch
  PROP (a ≺ b) ∧ (b ≺ c) ⟹ (a ≺ c)       −− transitiv
  DEF (a ⪯ b) = (a ≺ b ∨ (a = b)          −− Default-Definition
  DEF (a ≻ b) = (b ≺ a)                   −− Default-Definition
  DEF (a ⪰ b) = (b ⪯ a)                   −− Default-Definition
}
TYPECLASS Ord = Order.α
SPECIFICATION Order = {
  EXTEND PartialOrder
  PROP (a ≺ b) ∨ (a = b) ∨ (a ≻ b)        −− alle Elemente sind vergleichbar
}
```

9.6.3 Halbgruppen, Monoide *and all that*

Bei den zahlartigen Strukturen hatten wir viele Gesetze wie Assoziativität, Kommutativität etc. Diese lassen sich auch modularisiert spezifizieren. Programm 9.9 enthält zwei einfache Beispiele.

Mit diesen Spezifikationen lassen sich die zahlartigen Strukturen typisieren. Aber auch die Konkatenation von Sequenzen fällt unter diese Rubrik:

```
Integer : Monoid RENAMING (α, ∘, unit) AS (Int, +, 0)
Integer : Monoid RENAMING (α, ∘, unit) AS (Int, ·, 1)
Real : Monoid RENAMING (α, ∘, unit) AS (Real, +, 0)
Real : Monoid RENAMING (α, ∘, unit) AS (Real, ·, 1)
Sequence α : Monoid RENAMING (α, ∘, unit) AS (Seq, ++, ◊)
...
```

Wie man hier sieht, kann dieselbe Struktur eine Spezifikation unter verschiedenen Renaming-Morphismen erfüllen. (Darauf kommen wir gleich in Abschnitt 9.8 unter dem Stichwort „Views“ noch einmal zurück.)

Auf analoge Weise ließen sich weitere Standardstrukturen der Mathematik spezifizieren: Gruppen, kommutative Gruppen, Ringe, Körper, Vektorräume, Verbände etc.

Programm 9.9 Die Spezifikation von Halbgruppen und Monoiden

```
TYPECLASS Semigroup = Semigroup.α
SPECIFICATION Semigroup = {
  TYPE α
  FUN _ ∘ _ : α × α → α
  PROP (a ∘ b) ∘ c = a ∘ (b ∘ c)          -- assoziativ
}
TYPECLASS Monoid = Monoid.α
SPECIFICATION Monoid = {
  EXTEND Semigroup
  FUN unit: α
  PROP (unit ∘ a = a) ∧ (a ∘ unit = a)
}
```

9.6.4 Druckbares, Speicherbares ...

Für nahezu alle Arten von Werten – wieder mit Ausnahme der Funktionen – gibt es eine Repräsentation als Strings, so dass eine Ausgabe auf Druckern oder Bildschirmen ermöglicht wird.[8] Der Tradition von HASKELL folgend bezeichnen wir diese Spezifikation als *Show* und die umgekehrte Richtung als *Read*. (Allerdings ist die Definition in HASKELL etwas aufwendiger als unsere vereinfachte Darstellung.)

Programm 9.10 Die Signatur von *Show* und *Read*

```
TYPECLASS Show = Show.α
SIGNATURE Show = {
  TYPE α
  FUN show: α → String
}
TYPECLASS Read = Read.α
SIGNATURE Read = {
  TYPE α
  FUN read: String → α
}
```

Wir können uns hier mit den Signaturen begnügen, weil es keine allgemeinen Eigenschaften gibt, die man für diese Konversionsfunktionen ange-

[8] Auf der gleichen Linie liegt übrigens das Interface `Serializable` in JAVA. Damit werden alle Typen (d. h. Klassen) beschrieben, deren Werte (d. h. Objekte) sich in Dateien speichern und wieder laden lassen.

ben könnte. Es wäre zwar schön, wenn gelten würde $read \circ show = id$ und $show \circ read = id$, aber z. B. bei Gleitpunktzahlen können dem unter Umständen Rundungsprobleme entgegenstehen. Man kann auch nicht für alle beliebigen Datentypen fordern, dass die String-Repräsentation eindeutig parsierbar ist; denn damit würde man sich manchmal die Chance verbauen, „schöne“ Ausgaben zu erzeugen.

9.7 Subklassen

Wir verwenden hier Signaturen, Spezifikationen und Typklassen zur Typisierung von Strukturen und Typen, also analog zu klassischen Typen – nur eine Stufe höher (vgl. Abbildung 9.1 am Anfang dieses Kapitels). Deshalb liegt es nahe, auch die Idee der Subtypen auf diese Stufe zu heben.

Wir übernehmen deshalb die Begriffe und Notationen von Kapitel 7 in analoger Form auch für die nächsthöhere Ebene der Spezifikationen. Dies induziert dann auch eine entsprechende Relation zwischen den Typklassen. Beispiele:

$Order \subseteq OrderSig$	
$Equality \subseteq EqualitySig$	
$Order \subseteq PartialOrder \subseteq Equality$	$Ord \subseteq POrd \subseteq Eq$
$Monoid \subseteq Semigroup$	$Monoid \subseteq Semigroup$
$Number \subseteq Order$ RENAMING ...	$Numeric \subseteq Ord$
$Number \subseteq Monoid$ RENAMING ...	$Numeric \subseteq Monoid$
...	

Hier machen wir intensiven Gebrauch von der Feststellung in Abschnitt 7.3, dass der Subtyp einer Gruppe *mehr* Komponenten haben darf. Das heißt, die obigen Beziehungen beruhen auf zwei Arten von Einschränkungen: zum Beispiel bei ($PartialOrder \subseteq Equality$) sind es die zusätzlichen Komponenten, die den Subtyp ausmachen, während es bei ($Order \subseteq PartialOrder$) die zusätzlichen Constraints sind.

Anmerkung: Im Ansatz der Algebraischen Spezifikation arbeitet man ausschließlich mit Spezifikationen, im Sinne der Formalen Logik also mit (Repräsentationen von) Theorien. Die Semantik dieser Theorien sind Mengen von Modellen, genauer: Mengen von Algebren. Ein zentraler Begriff ist dann die so genannte Implementierungsrelation. Eine Spezifikation A implementiert eine Spezifikation B, wenn alle Modelle von A auch Modelle von B sind. Wenn wir das als (A REALIZES *B) notieren, dann können wir z. B. sowohl (Integer* REALIZES *Order) schreiben als auch (Order* REALIZES *Equality). Der Grund für diese Homogenität ist, dass in dem algebraischen Ansatz eine Struktur wie Integer auch nur eine Spezifikation ist, allerdings eine Spezifikation mit einelementiger Modellmenge (bis auf Isomorphie).*

In unserer funktionalen Programmierwelt sind die Strukturen keine entarteten Spezifikationen, sondern die Modelle selbst. Deshalb haben wir zwischen den Strukturen und den Spezifikationen eine Typisierungsrelation, während zwischen den Spezifikationen eine Subtyp-Relation besteht (die dem REALIZES *der Algebra entspricht).*

Diese Skizze des algebraischen Ansatzes soll hier genügen. Für detaillierte Darstellungen verweisen wir auf die Literatur, wobei [19, 101] einen sehr guten Zugang liefern.

9.8 Views und Mehrfachtypisierung

Bei Signaturen und Spezifikationen tritt ein Phänomen sehr häufig auf, das bei klassischen Typen zwar auch denkbar ist, aber eher exotisch wirkt: *mehrfache Typisierung* (vgl. Abschnitt 6.1.7).

9.8.1 Mehrfachtypisierung bei Typklassen

Schon in den wenigen und einfachen Beispielen dieses Kapitels haben wir Situationen wie

Integer : *Order* RENAMING ...
Integer : *Monoid* RENAMING ...
Integer : *Number* RENAMING ...
Integer : *Show* RENAMING ...

Das heißt, die Struktur *Integer* ist eine Instanz von vielen Spezifikationen. Dies ist aus softwaretechnischer Sicht wesentlich: Man braucht die Möglichkeit der Mehrfachtypisierung. Das zeigt sich auch in objektorientierten Sprachen wie JAVA, in denen eine Klasse viele Interfaces implementieren kann.

Damit stellt sich aber die Frage, wo und wie man diese Eigenschaften aufschreibt. Die sauberste Lösung, die auch keine weiteren Ad-hoc-Notationen braucht, basiert auf dem generellen Gruppenmechanismus und Propertys und ist im Programm 9.11 skizziert.

Programm 9.11 Package *Arithmetic* mit Typisierungen

```
PACKAGE Arithmetic = {
  STRUCTURE Integer = { ... }
  PROP Integer : Order RENAMING ...
  PROP Integer : Monoid RENAMING ...
  PROP Integer : Number RENAMING ...
  PROP Integer : Show RENAMING ...
  STRUCTURE Real = { ... }
  PROP Real : Order RENAMING ...
  PROP Real : Monoid RENAMING ...
  ...
}
```

Wenn man allerdings Sprachen wie JAVA ansieht, dann gibt es eine Tendenz, solche Eigenschaften jeweils lokal zu der Struktur hinzuzufügen. Das geht nur über den Weg eines neuen Schlüsselwortes wie REALIZES (oder `implements` in JAVA) oder über einen Operator, der die Konjunktion von Typen erlaubt. Wir skizzieren hier die letztere Variante.

```
STRUCTURE Integer :     (Order RENAMING ...)
                      & (Monoid RENAMING ...)
                      & (Number RENAMING ...)
                      & (Show RENAMING ...)     = {
  TYPE Int = GIVEN
  FUN _ + _ = ...
  ...
}
```

Man sieht deutlich, dass die Version in Programm 9.11 die klarere und besser lesbare ist. Der Grund dafür ist ganz einfach: Wir respektieren hier die softwaretechnische Forderung, Programme aus wohl definierten und sauber abgegrenzten Komponenten aufzubauen. Demgegenüber nehmen funktionale Programmiersprachen (ebenso objektorientierte Sprachen wie JAVA) gerne einen etwas altbackenen Standpunkt ein, in dem man viele einzelne Deklarationen einfach nebeneinander hinschreibt.

Dabei ist die Frage aus konzeptueller Sicht absolut klar: Eigenschaften wie „*Integer* ist eine Instanz von *Order*“ sind Aussagen *über* die Struktur und sollten deshalb nicht *in* die Struktur hineingeschrieben werden.

9.8.2 Views (Mehrfache Sichten)

Bisher haben wir den einfachen Fall betrachtet, dass eine Struktur Instanz mehrerer Spezifikationen sein kann und damit der zugehörige Typ Instanz mehrerer Typklassen. Eine weitere Komplikation entsteht dadurch, dass eine Struktur *auf mehrere Weisen* Instanz der gleichen Spezifikation sein kann. Wir illustrieren dieses Phänomen anhand eines einfachen Beispiels.

Eine wesentliche Motivation für Typklassen haben wir am Anfang dieses Kapitels aus dem Problem des Sortierens bezogen. Denn wir mussten die Sequenzen auf Elementtypen beschränken, die eine Vergleichsoperation besitzen. Das löst aber nur einen Teil der Probleme, wie wir in Szenarien folgender Bauart sehen können: *Es sei eine Menge von Kunden als Sequenz gegeben. Man sortiere sie zuerst nach Kundennummer, dann nach Namen und schließlich nach Umsatz.* Ähnliche Szenarien finden sich bei Algorithmen für Punktmengen, bei denen die Punkte einmal nach der x-, ein anderes Mal nach der y-Komponente und ein drittes Mal nach dem Winkel sortiert werden sollen.

Wir diskutieren diese Szenarien an einem besonders einfachen, aber dennoch praktisch relevanten Beispiel: *Wir wollen Zahlen sowohl auf- als auch absteigend sortieren.* Die Grundlage stellen die folgenden Propertys dar:

PROP *Integer*: *Order* RENAMING $(\alpha, \prec, \preceq, \ldots)$ AS $(Int, <, \leq, \ldots)$
PROP *Integer*: *Order* RENAMING $(\alpha, \prec, \preceq, \ldots)$ AS $(Int, >, \geq, \ldots)$

Wie man hier sieht, wird die Struktur *Integer* auf zwei unterschiedliche Weisen zur Instanz von *Order*.

Mit Hilfe dieser Funktionen soll jetzt eine Sequenz von Zahlen sowohl in auf- als auch in absteigender Reihenfolge sortiert werden. Klassischerweise hat man für diese Art von Aufgaben zwei Lösungsansätze zur Verfügung:

- Die Standardlösung besteht darin, der Sortierfunktion die Vergleichsoperatoren als zusätzliche Parameter mitzugeben:

  FUN $sort[\alpha]$: $(\alpha \times \alpha \to Bool) \times (\alpha \times \alpha \to Bool) \times (\alpha \times \alpha \to Bool)$
  $\to Seq\ \alpha \to Seq\ \alpha$
  DEF $sort[\alpha](<,=,>)(s) = \ldots\ sort[\alpha](<,=,>)(s')\ \ldots$

  Diese Lösung ist in höchstem Maße unbefriedigend, weil sowohl in den rekursiven Aufrufen als auch in den Applikationen von *sort* eine Fülle von Argumenten anzugeben sind:

  $\ldots sort[Int](<, =, >)(list) \ldots$
  $\ldots sort[Int](>, =, <)(list) \ldots$

- In Sprachen mit generischen Strukturen kann man diesen Overhead etwas reduzieren, indem man die Vergleichsoperatoren nicht zu Parametern der Funktion macht, sondern zu Parametern der umfassenden Struktur:

  STRUCTURE $Sorting[\alpha: \mathit{Ord}](<: \alpha \times \alpha \to Bool,$
  $=: \alpha \times \alpha \to Bool,$
  $>: \alpha \times \alpha \to Bool\ ) = \{$
  FUN $sort: Seq\ \alpha \to Seq\ \alpha$
  DEF $sort\ s = \ldots$
  }

  Aber an der Anwendungsstelle muss noch viel Aufwand getrieben werden, um die richtigen Instanzen der generischen Strukturen zu importieren:

  IMPORT $Sort_{up} = Sorting[Int](<,=,>)$
  IMPORT $Sort_{down} = Sorting[Int](>,=,<)$

  $\ldots Sort_{up}.sort(list) \ldots$
  $\ldots Sort_{down}.sort(list) \ldots$

  Dieser Aufwand macht sich eigentlich nur bezahlt, wenn man an der Anwendungsstelle viele Applikationen der Sortierfunktion hat – was eher selten der Fall sein dürfte.

Fazit: So richtig elegant ist keine dieser beiden Lösungen.

Beide Lösungen leiden am gleichen methodischen Defizit. Sie versuchen zwar korrekterweise, der Struktur *Integer* zwei verschiedene Ordnungen aufzuprägen, aber sie tun das an der Applikationsstelle statt an der Definitionsstelle. Und damit wird das Sortieren belastet.

Eine bessere Lösung entsteht, indem man die unterschiedlichen Ordnungen für *Integer* schon an der Definitionsstelle von *Integer* selbst angibt und nicht erst an der Applikationsstelle, also beim Sortieren.

- *Erster Schritt.* Wir geben den beiden unterschiedlichen Instanzen explizite Namen:

  SPECIFICATION $Ascending =$
  $\quad Order$ RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, <, \ldots)$
  SPECIFICATION $Descending =$
  $\quad Order$ RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, >, \ldots)$
  TYPECLASS $Asc = Ascending.Int$
  TYPECLASS $Desc = Descending.Int$

  Man beachte, dass – rein formal gesehen – Int hier noch immer eine existenzielle Typvariable ist; aber sie hat für den Compiler schon den passenden Namen.
- *Zweiter Schritt.* Wir rufen die Sortierfunktion mit dem entsprechenden Casting auf:

  $\ldots\ sort[Int\colon Desc](list)\ \ldots$

Es bietet sich an, ein solches Konzept auch syntaktisch zu unterstützen. Bevor wir uns aber mit der Frage des Sprachdesigns befassen, müssen wir zeigen, dass diese Vorgehensweise tatsächlich den gewünschten Effekt hat. Dazu verwenden wir wieder unsere speziellen Annotationen, mit denen wir die Laufzeittypen illustrieren. Als Erstes müssen wir sehen, dass sich eine Definition wie

FUN $sort\colon\ [\alpha\colon Ord] \mapsto Seq\ \alpha^{\langle Ord\rangle} \to Seq\ \alpha^{\langle Ord\rangle}$
DEF $sort[\alpha](list) = \ldots\ \prec^{\langle \alpha\times\alpha\to Bool\rangle}\ \ldots$

eigentlich nicht auf eine Typvariable α der Typklasse Ord bezieht, sondern auf eine entsprechende Strukturvariable S der Spezifikation *Order*:

FUN $sort\colon\ [S\colon Order] \mapsto Seq(S^{\langle Order\rangle}.\alpha) \to Seq(S^{\langle Order\rangle}.\alpha)$
DEF $sort[S](list) = \ldots\ S^{\langle Order\rangle}.\prec^{\langle S.\alpha\times S.\alpha\to Bool\rangle}\ \ldots$

Bei der Applikation erfolgt dementsprechend eine Instanziierung mit der ganzen Struktur. Das heißt, ein Aufruf der Art

$\ldots\ sort[Int\colon Desc](list)\ \ldots$

entspricht tatsächlich einem Aufruf der Art

$\ldots\ sort[Integer\colon Descending](list)\ \ldots$

Dies ist die lesbare Form, die vom Programmierer geschrieben wird. Für unsere Diskussion müssen wir allerdings die diversen Castings explizit machen. Damit ergibt sich insgesamt folgende Situation:

$\ldots\ sort[(Integer$ AS $Descending)$ AS $Order](list)\ \ldots$

Zur besseren Lesbarkeit kürzen wir das aufwendige Casting ab:

$$\Psi(Integer) \stackrel{\text{def}}{=} (Integer \text{ AS } Descending) \text{ AS } Order$$

Jetzt müssen wir prüfen, welchen Typ diese Instanz von *sort* hat und wie die relevanten Operatoren im Rumpf erkannt werden. Der Typ ergibt sich folgendermaßen:

$$\ldots\ sort[\Psi(Integer)]^{\langle Seq[\Psi(Integer).\alpha] \to Seq[\ldots]\rangle}\ \ldots$$

Bei der Auswertung dieses Aufrufs wird im Rumpf der Vergleichsoperator folgendermaßen annotiert:

$$\ldots\ \Psi(Integer).\prec^{\langle \Psi(Integer).\alpha \times \Psi(Integer).\alpha \to Bool\rangle}\ \ldots$$

Damit hier die korrekten Interpretationen erfolgen, müssen folgende Gleichheiten gelten:

$$\Psi(Integer).\alpha \stackrel{!}{=} Integer.Int$$
$$\Psi(Integer).\prec \stackrel{!}{=} Integer.>$$
$$\ldots$$

Wenn wir für Ψ wieder die Definition einsetzen, ergibt sich daraus:

$$((Integer \text{ AS } Descending) \text{ AS } Order).\alpha \stackrel{!}{=} Integer.Int$$
$$((Integer \text{ AS } Descending) \text{ AS } Order).\prec \stackrel{!}{=} Integer.>$$
$$\ldots$$

Wenn wir den Ausdruck ((*Integer* AS *Descending*) AS *Order*) genauer analysieren, dann ergibt sich als Erstes ein Casting von *Integer* zu *Descending*, was wegen der passenden Selektornamen unproblematisch ist. *Descending* ist seinerseits nur durch einen Renaming-Morphismus aus *Order* hervorgegangen, ist also von der Form $\Phi(Order)$. Damit entsteht die Zwischenform

$$Integer^{\langle \Phi(Order)\rangle} \text{ AS } Order$$

Dieses Casting muss dazu führen, dass unter den Selektornamen α und $\prec$ von *Order* die entsprechenden Elemente von *Integer* erreicht werden. Das bedeutet, dass auf *Integer* das inverse Renaming $\bar{\Phi}$ anzuwenden ist.

Diese Rechnung zeigt, dass das Konzept in der Tat funkioniert, auch wenn der Compiler einiges an Arbeit zu leisten hat.

Wir wollen für das hier geschilderte Konzept eine explizite syntaktische Kennzeichnung einführen. Dies ist zwar nicht zwingend notwendig, weil unsere bisherigen Schlüsselwörter (wie gerade gezeigt) ausreichen, aber es unterstützt in den entsprechenden Applikationen die Intuition.

Definition (View)

Wenn eine Struktur auf mehrere Weisen Instanz einer Signatur oder Spezifikation sein kann, dann sprechen wir von ***Views***. Dafür benutzen wir folgende syntaktische Darstellung:

VIEW *View* = *Spec* RENAMING ...

Sollte zu *Spec* eine Typklasse *Class* assoziiert sein, dann dürfen wir das Renaming auch direkt auf die Typklasse anwenden:

VIEW *View* = *Class* RENAMING ...

Dies induziert dann das entsprechende Renaming auf der Spezifikation.

Wenn für einen gegebenen Typ bzw. für die zugehörige Struktur ein Casting in einen solchen View erfolgt, dann wird de facto das Renaming invertiert, so dass die Items der Struktur unter den passenden Namen erreichbar sind.

Damit haben wir zwei gleichwertige Möglichkeiten, um solche Instanziierungen anzugeben. Mit den „normalen“ Schlüsselwörtern erhalten wir folgendes Programm:

STRUCTURE *Integer* = { ... }

PROP *Integer*: *Order* RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, <, \ldots)$
PROP *Integer*: *Order* RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, >, \ldots)$

Mit dem Schlüsselwort VIEW können wir die beiden Instanzen – also im Wesentlichen die Renaming-Morphismen – benennen:

STRUCTURE *Integer* = { ... }

VIEW *Ascending* = *Order* RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, <, \ldots)$
VIEW *Descending* = *Order* RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, >, \ldots)$

PROP *Integer*: *Ascending*
PROP *Integer*: *Descending*

Zusätzlich – oder alternativ dazu – können wir das Renaming auch direkt für die Typklassen angeben, sodass wir keinen eigene Namen für die umbenannten Spezifikationen brauchen.

STRUCTURE *Integer* = { ... }

VIEW *Asc* = *Ord* RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, <, \ldots)$
VIEW *Desc* = *Ord* RENAMING $(\alpha, \prec, \ldots)$ AS $(Int, >, \ldots)$

PROP *Int*: *Asc*
PROP *Int*: *Desc*

Diese Lösung hat gegenüber unserem früheren Versuch mit generischen Strukturen zwei große Vorteile:

- Man kann an beliebigen Stellen im Programm eine Applikation der Art
  ... *sort*[*Int*: *Desc*](*list*) ...

schreiben, ohne das Sortierprogramm und seine Applikationen umständlich in eine generische Struktur und passend instanziierte Importe einpacken zu müssen.

- Vor allem aber wird aus softwaretechnischer Sicht das Design klarer: Jetzt steht die Information, dass die Struktur *Integer* auf verschiedene Weisen eine Instanz von *Order* ist, dort, wo sie hingehört, nämlich bei der Struktur selbst.

Wir werden wichige Beispiele für dieses Konzept in Kapitel 10 sehen, wo wir Fixpunktalgorithmen über so genannten CPOs entwickeln. Dabei wird sich zeigen, dass Datenstrukturen auf sehr unterschiedliche Weise zu CPOs werden können.

9.9 Beispiel: Physikalische Dimensionen

Die mit den Typklassen gewonnene Ausdrucksmächtigkeit hat nicht nur theoretische, sondern auch praktische Relevanz. Das lässt sich mit einem Beispiel illustrieren, das ein altbekanntes Defizit nahezu aller Programmiersprachen behebt.[9]

Physiker und Ingenieure rechnen aus gutem Grund mit dimensionsbehafteten Größen. Eine physikalische Notation wie $12.5\frac{m}{sec^2}$ hat eben eine viel größere Aussagekraft als die bloße reelle Zahl 12.5. Das gilt erst recht im Kontext von Gleichungen wie $25\frac{m}{sec}/2sec = 12.5\frac{m}{sec^2}$. Diese Art von zusätzlicher Typsicherheit wäre natürlich auch in der Programmierung physikalisch-technischer Applikationen ein gewaltiger Vorteil. Leider reichen aber die klassischen Sprachmittel der gängigen Programmiersprachen dafür nicht aus.

Aber mit dem Mittel der Typklassen wird es möglich, solche Konzepte völlig flexibel in Programme – oder auch Bibliotheken – einzubauen. Wir illustrieren das am einfachsten Beispiel dimensionierter Größen: Weg – Geschwindigkeit – Beschleunigung (s. Programm 9.12).

Man beachte, dass die Typklassen *Base* und *Dim* hier nicht über Signaturen oder Spezifikationen induziert, sondern mittels der üblichen Operatoren Aufzählung und Tupelbildung eingeführt werden (s. Abschnitt 9.1).

Wir diskutieren die Elemente, die in diesem Programm verwendet werden, der Reihe nach, wobei wir zum leichteren Lesen jeweils die Zeilennummern angeben.

1 Wir versuchen gar nicht erst, ein neues Schlüsselwort für diese Kollektion von Programmelementen zu erfinden; deshalb bleiben wir beim generischen Begriff GROUP.

[9] In ADA ist das Defizit zwar erkannt, aber nur partiell gelöst worden. Denn es gibt nur eine fixierte, fest in den Compiler eingebaute Liste von Dimensionstypen.

Programm 9.12 Die Gruppe *Dimension*

```
GROUP Dimension = {                                                    1
  PRIVATE TYPECLASS Base = { Real }                                    2
  TYPECLASS Dim = ( base = Base, m = Int, s = Int )                    3
  TYPE M: Dim = ( Real, 1, 0 ) ⊆ ( Real )                              4
  TYPE S: Dim = ( Real, 0, 1 ) ⊆ ( Real )                              5
  TYPE _ · _: Dim × Dim → Dim                                          6
  DEF (Real, m1, s1) · (Real, m2, s2) = ( Real, m1 + m2, s1 + s2 )     7
  TYPE _ / _: Dim × Dim → Dim                                          8
  DEF (Real, m1, s1) / (Real, m2, s2) = ( Real, m1 − m2, s1 − s2 )     9
  TYPE _ ^ _: Dim × Int → Dim                                          10
  DEF (Real, m, s) ^ i = ( Real, m · i, s · i)                         11
  FUN _m: Real → M                                                     12
  DEF x m = (x) AS M                                                   13
  FUN _s: Real → S                                                     14
  DEF x s = (x) AS S                                                   15
  FUN add: [δ: Dim] ↦ δ × δ → δ                                        16
  DEF add[δ](x, y) = ( (x AS Real) + (y AS Real) ) AS δ                17
  FUN mult: [δ1: Dim, δ2: Dim] ↦ δ1 × δ2 → δ1 · δ2                     18
  DEF mult[δ1, δ2](x, y) = ( (x AS Real) · (y AS Real) ) AS δ1 · δ2    19
  FUN div: [δ1: Dim, δ2: Dim] ↦ δ1 × δ2 → δ1 · δ2                      20
  DEF div[δ1, δ2](x, y) = ( (x AS Real) · (y AS Real) ) AS δ1 / δ2     21
  FUN pow: [δ: Dim] ↦ δ × (i: Int) ↦ δ ^ i                             22
  DEF pow[δ](x, i) = ( (x AS Real) ^ i ) AS δ ^ i                      23
}                                                                      24
```

2 Alle dimensionierten Typen sind „Dekorationen“ des Typs *Real*. Deshalb führen wir eine – einelementige – Typklasse *Base* ein, die genau diesen Typ *Real* enthält.

3 Der interessanteste Teil des Programms ist die Typklasse *Dim* selbst. Die Elemente dieser Klasse sind Tripel, bestehend aus dem Basistyp *Real* und den Exponenten der Meter-Komponente und der Sekunden-Komponente. Wenn also ein Physiker schreibt $12\frac{m}{s^2}$, dann ergibt sich hier der Dimensionstyp $(Real, 1, -2)$: *Dim*. Man beachte, dass wir „gemischte“ Tupel haben, bestehend aus einem Typ und zwei Werten.
Übrigens: Auch wenn der Basistyp immer der gleiche ist, nämlich *Real*, sollte man ihn aus Gründen der Klarheit nicht weglassen. Wir werden sogar gleich sehen, dass wir ihn essenziell benötigen.

4-5 Wir führen zwei spezielle Dimensionstypen ein, *M* für Meter und *S* für Sekunden. Das geschieht, indem jeweils der betreffende Exponent auf 1 gesetzt wird, der andere auf 0.

Wir merken explizit an, dass beide Typen Subtypen des Eintupels (*Real*) sind (vgl. Kapitel 7). Das ist wichtig, um entsprechende Castings vornehmen zu können.

`6-9` Die physikalischen Konstruktionen sm oder $\frac{m}{s}$ brauchen Gegenstücke in unserem Design. Wir können das schreiben als $S \cdot M$ bzw. $M \,/\, S$. Die Implementierung besteht einfach darin, dass die Indizes addiert bzw. subtrahiert werden.

`10-11` Auch eine Schreibweise wie m^2 braucht ein Gegenstück in unserem Design. Wir schreiben das als $m \,\hat{}\, 2$. Die Realisierung von $\_ \,\hat{}\, i$ besteht einfach darin, alle Exponenten mit i zu multiplizieren.

`12-15` Es ist bequem, Konstanten wie etwa 1 m oder 2 s genau so schreiben zu können. Dazu werden zwei entsprechende Postfix-Operatoren eingeführt. Die Implementierung ist ein bisschen trickreich; wir passen den Wert (x) vom einelementigen Tupeltyp (*Real*) auf den Supertyp M bzw. S an (gemäß den Subtyp-Relationen aus Zeile 4-5).

`16-17` Addieren ist nur möglich, wenn die Dimensionen identisch sind, also z. B. $2m^2 + 3m^2$, aber nicht $2m + 4s$. Genau das wird in der Funktionalität von *add* ausgesagt. Die Funktion ist polymorph, wobei die Typvariable δ auf Typen der Klasse *Dim* beschränkt wird.
Man beachte, dass auch in der Definition die Typvariable δ vorkommt, weil sie beim Casting benötigt wird.

`18-21` Multiplikation und Division sind polymorph in zwei Variablen δ_1 und δ_2. Der Resultattyp ergibt sich dabei gerade aus den entsprechenden Typoperatoren aus Zeilen 6-9. So wird z. B. aus $\frac{6:M}{2:S \,\hat{}\, 2}$ der Wert $3:(M \,/\, S \,\hat{}\, 2)$.

`22-23` Die Typisierung der Operation *pow* hängt nicht nur von der Typvariablen δ ab, sondern auch vom Parameter*wert* i. Wir haben es hier also mit einem abhängigen Typ zu tun, wie wir ihn schon aus Abschnitt 8.2 von Beispielen wie $FixedList[n][\alpha]$ her kennen.

Man beachte, dass die Typen M und S Subtypen des Typs *Real* sind, genauer, des Eintupels, das nur aus *Real* besteht. Damit sind Upcasts der Art $\_\,\text{AS}\ Real\colon\ M \to Real$ problemlos möglich: $1.5^{\langle Real,1,0\rangle}\,\text{AS}\ Real \rightsquigarrow 1.5^{\langle Real\rangle}$. Bei Downcasts ist der Compiler stärker gefordert, er muss nämlich erkennen, dass die beiden fehlenden Komponenten jeweils Konstanten sind und damit automatisch ergänzt werden können: $1.5^{\langle Real\rangle}\,\text{AS}\ M \rightsquigarrow 1.5^{\langle Real,1,0\rangle}$.

10

Beispiel: Berechnung von Fixpunkten

Wer einmal sich selbst gefunden, der kann
nichts auf dieser Welt mehr verlieren.
Stefan Zweig

In diesem Kapitel wollen wir die Verwendung von Typklassen an einer wichtigen Gruppe von Programmieraufgaben illustrieren. Häufig muss man Gleichungssysteme folgender Bauart lösen:

$$x = E[x] \tag{10.1}$$

Damit eng verwandt ist das Problem, den transitiven Abschluss einer Relation zu bilden. Programme, die die Lösung solcher Gleichungen konstruktiv berechnen, basieren auf dem Prinzip der *Fixpunktbildung*. Dieses Prinzip lässt sich – einschließlich vieler Optimierungen – sehr kompakt und allgemein anwendbar in Form funktionaler Programme realisieren.[1]

Wir beginnen mit einem Beispiel und stellen dann die Grundidee der Fixpunktberechnung dar. Auf dieser Basis bauen wir weiter auf, um auch die algorithmische Lösung von Gleichungssystemen systematisch zu realisieren.

10.1 Beispiel: Erreichbarkeit in einem Graphen

Wenn man zu einem gegebenen Knoten in einem gerichteten Graphen wissen will, welche Knoten von ihm aus zu erreichen sind, muss man den transitiven Abschluss der Nachfolgerrelation bilden.

Wir betrachten also einen gerichteten Graphen G. Seine *charakteristische Funktion* ordnet jedem Knoten seine Nachbarknoten bzw. „Nachfolger“ zu:

FUN *succ*: *Graph* → *Node* → *Set Node*

[1] Unsere Darstellung orientiert sich stark an Arbeiten von R. Paige und J. Cai [28, 29]. Wer mehr über die Mathematik nachlesen will, kann z. B. in [146] eine gute Übersicht finden.

Mit Hilfe des „.“-Operators aus Abschnitt 1.2.1 schreiben wir $G.succ(x)$ für die Menge der Nachfolgerkanten von x im Graphen G.

Auf der Basis dieser charakteristischen Funktion spezifiziert Programm 10.1 die Menge der in einem Graphen G von einem Knoten x aus erreichbaren Knoten. (Wir müssen das Schlüsselwort PROP anstelle von DEF verwenden, weil die rechte Seite das nicht-ausführbare Sprachmittel LEAST benutzt.)

Programm 10.1 Spezifikation der erreichbaren Knoten

```
SPECIFICATION Reachability = {
  FUN reachable: Graph → Node → Set Node
  PROP reachable(G)(x) = LEAST S • x ∈ S ∧ succs(S) ⊆ S
                                 WHERE succs(S) = ⋃_{x∈S} G.succ(x)
}
```

Mit LEAST drücken wir aus, dass wir die kleinste Menge mit der betreffenden Eigenschaft suchen. Die Funktion $succs(S)$ berechnet die „Vereinigung der Nachfolger aller Knoten in S “.[2]

Damit die Techniken, die wir im weiteren Verlauf erarbeiten werden, anwendbar sind, müssen zwei kleine Adaptionen erfolgen:

1. Statt $x \in S$ werden wir $\{x\} \subseteq S$ schreiben, damit die Typisierung stimmt.
2. Das Prädikat $succs(S) \subseteq S$ schreiben wir in die (äquivalente) Form einer Gleichung um: $S = S \cup succs(S)$.

Nach diesen Anpassungen sind die Techniken der folgenden Abschnitte anwendbar. Das Ergebnis ist dann der konkrete Algorithmus in Programm 10.2.

Programm 10.2 Algorithmische Berechnung der erreichbaren Knoten

```
STRUCTURE Reachable = {
  USE Fixpoint(Set Node)
  DEF reachable(G)(x) = fixpoint(succs)({x})
                        WHERE succs(S) = S ∪ ⋃_{x∈S} G.succ(x)
}
```

In diesem Programm verwenden wir die – im weiteren Verlauf des Kapitels zu entwickelnde – Struktur *Fixpoint*, die insbesondere eine Funktion *fixpoint* bereitstellt. Diese Struktur ist generisch und wird hier mit unserem gewünschten Ergebnistyp *Set Node* instanziiert.

[2] Wie in Kapitel 0 erläutert, bevorzugen wir den Lesekomfort mathematischer Schreibweisen. In der Notation klassischer funktionaler Programmiersprachen wird *succs* im Map-Reduce-Stil als $succs(S) = \cup\ /\ G.succ * S$ geschrieben.

10.2 Ein bisschen Mathematik: CPOs und Fixpunkte

Fixpunkte sind in der Mathematik schon lange bekannt. Wir skizzieren hier nur ganz kurz den *minimalen* Rahmen, den wir brauchen, um unsere Verfahren beschreiben zu können.

10.2.1 Vollständige partielle Ordnungen – CPOs

Wir arbeiten in der Programmierung an allen möglichen Stellen mit *Gleichungen* und *Fixpunkten*. Damit diese überhaupt sinnvoll sind, müssen sie in geeigneten Strukturen „leben“. Diese Strukturen wollen wir uns als erstes ansehen.[3]

Grundlegend ist der Begriff der *vollständigen partiellen Ordnung*, abgekürzt: *CPO* (vom englischen Begriff *complete partial order*).

Definition (vollständige partielle Ordnung (CPO))

Eine ***partielle Ordnung*** ist eine Menge A mit einer reflexiven, antisymmetrischen und transitiven Relation „$\preceq$“.

Die Operation $x \sqcup y$ (im Englischen *join* genannt) liefert das kleinste Element z mit $x \preceq z$ und $y \preceq z$ (sofern ein solches Element existiert). Die Verallgemeinerung von „$\sqcup$“ von zwei Elementen auf ganze Mengen X ist evident. Man nennt $\sqcup X$ (sofern es existiert) die ***kleinste obere Schranke*** oder auch das ***Supremum*** von X.

Eine (aufsteigende) ***Kette*** in einer partiellen Ordnung $(A, \preceq)$ ist eine (endliche oder unendliche) Teilmenge $K \subseteq A$, die aus streng wachsenden Elementen besteht: $K = \{x_0 \prec x_1 \prec x_2 \prec \dots\}$.

Eine partielle Ordnung $(A, \preceq)$ heißt ***vollständig***, wenn jede aufsteigende Kette $K \subseteq A$ eine kleinste obere Schranke $\sqcup K$ in A besitzt.

Eine solche ***CPO*** hat insbesondere auch ein *kleinstes Element*, das üblicherweise als $\bot$ („Bottom“) bezeichnet wird. Es ist eindeutig bestimmt als das Supremum der leeren Kette: $\bot = \sqcup \emptyset$.

Aus Terminierungsgründen ist für uns auch der – weniger klassische – Begriff der Kettenendlichkeit interessant: Eine partielle Ordnung $(A, \preceq)$ ist *kettenendlich*, wenn sie keine unendlichen Ketten besitzt.

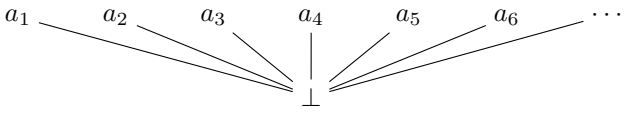
Wir nennen ein Element $a \in A$ *endlich*, wenn die Menge $\{x \in A \mid x \preceq a\}$ keine unendliche Kette enthält. Wir nennen a *endlich relativ zu* $w \in A$, wenn die Menge $\{x \in A \mid w \preceq x \preceq a\}$ keine unendliche Kette enthält.

CPOs erlauben das Arbeiten mit Fixpunkten. Bevor wir dies im Detail betrachten, wollen wir kurz die typischen Möglichkeiten erörtern, mit denen sich CPOs konstruieren lassen.

[3] Ursprünglich wurden alle folgenden Begriffe und Resultate in der Mathematik im Rahmen der Theorie der vollständigen Verbände erarbeitet (s. z. B. [23, 39]). In der Informatik hat man aber erkannt, dass sie sich auf die schwächere – und in der Informatik adäquatere – Theorie der CPOs übertragen lassen.

10.2.2 Standardkonstruktionen für CPOs

Die einfachste Form von CPOs sind die so genannten *flachen CPOs*: Zu einer Menge assoziiert man noch ein $\bot$-Element hinzu. Alle anderen Elemente sind miteinander unvergleichbar. Der polymorphe Typ *Maybe* α kann als flache CPO aufgefasst werden, wobei *fail* die Rolle von $\bot$ übernimmt.



Ein weiteres Beispiel liefert das *direkte Produkt* $A_1 \times \ldots \times A_n$ von CPOs A_i, das auf einfache Weise wieder zur CPO wird. Wir können dabei zwei Varianten unterscheiden – und diese beiden Varianten liefern genau den Unterschied zwischen strikten und nichtstrikten Funktionen (vgl. Abschnitt 1.3), und damit auch die Basis für unendliche (lazy) Listen und ähnliche Datenstrukturen (vgl. Kapitel 2).

- *Nichtstriktes Produkt*:

$$(a_1, \ldots, a_n) \preceq (b_1, \ldots, b_n) \iff \forall i : a_i \preceq b_i$$

- *Striktes Produkt* (auch „*Smash Produkt*“): Wie oben, aber speziell wird gesetzt

$$(a_1, \ldots, a_n) = \bot \iff \exists i : a_i = \bot$$

Analog kann man auch die *direkte Summe* bilden: $A + B$ ist im Wesentlichen die disjunkte Vereinigung der beiden CPOs mit ihren jeweiligen Ordnungen. Allerdings identifiziert man üblicherweise die beiden Bottom-Elemente: $\bot_A = \bot_B \stackrel{\text{def}}{=} \bot$.

Schließlich wird auch noch aus dem *Funktionsraum* eine CPO, sofern man sich auf so genannte stetige Funktionen – s. Abschnitt 10.2.4 – beschränkt, d. h. $[A \to B] = \{f : A \to B \mid f \textit{ ist stetig}\}$. Die Ordnung wird punktweise induziert: $f \preceq g \iff \forall x : f(x) \preceq g(x)$.

Anmerkung: In der denotationellen Semantik (s. Abschnitt 1.3), die auf Dana Scott und Christopher Strachey zurückgeht, wird die Bedeutung von Programmen mit Hilfe spezieller CPOs erklärt (die manchmal als Scott domains *bezeichnet werden). Dabei steht das $\bot$ für „undefiniert“, also für nichtterminierende Programme. Tupelbildung etc. werden definiert wie oben beschrieben, wobei man zwischen strikten und nichtstrikten Sprachkonstrukten wählen kann. Dann lässt sich zeigen, dass alle üblichen Sprachkonstrukte – if-then-else, Funktionsdefinition und -applikation, Schleifen, Variablenzuweisungen usw. – stetige Funktionen sind. (Näheres dazu findet man z. B. in [146, 126]).*

10.2.3 CPO-Konstruktion durch Idealvervollständigung

In der Analysis erfährt man, wie aus den rationalen Zahlen die reellen gebildet werden. Die Idee ist grob die folgende: Wir können eine irrationale Zahl nicht wirklich erzeugen. Aber wenn sie sich auf systematische Weise beliebig annähern lässt, dann „kennen wir sie gut genug“. Also identifizieren wir die (fiktive) irrationale Zahl einfach mit der Menge aller ihrer Annäherungen – und davon können wir so viele konkret anfassen, wie wir wollen.

Genau so können wir es auch mit unendlichen Elementen machen. Sie lassen sich nicht wirklich konstruieren, aber wir können sie so genau annähern, wie wir wollen. Diese Philosophie führt auf das Konzept der so genannten *Idealvervollständigung*.

Definition (gerichtete Menge, Kegel, Ideal)

Wir betrachten partielle Ordnungen. Eine Menge D ist eine ***gerichtete Menge*** (engl.: *directed set*), wenn jede *endliche* Teilmenge $E \subseteq D$ ein Supremum in D besitzt: $\exists \hat{e} \in D : \hat{e} = \sqcup E$. Beachte: Eine gerichtete Menge D kann nicht leer sein; denn wegen $\emptyset \subseteq D$ gilt $\bot \stackrel{\text{def}}{=} \sqcup \emptyset \in D$.

Eine Menge K ist ein ***Kegel***, wenn sie nach unten abgeschlossen ist; d. h., mit jedem Element x sind auch alle kleineren Elemente in K enthalten, d. h. $x \in K \wedge y \preceq x \implies y \in K$.

Ein ***Ideal*** ist ein gerichteter Kegel; d. h., es ist nach unten abgeschlossen *und* jede endliche Teilmenge hat ein Supremum.

Ideale liefern eine weitere Möglichkeit, CPOs zu konstruieren. Denn es gilt folgender Satz (hier ohne Beweis):

Theorem 10.1 (Idealvervollständigung). *Sei $(P, \preceq)$ eine partielle Ordnung; sei ferner I_P die Menge aller Ideale von P. Dann gilt: $(I_P, \subseteq)$ ist eine CPO, die P „vervollständigt“.*

„Vervollständigt“ heißt: Jedes Element $x \in P$ wird mit „seinem“ Ideal $I_x \stackrel{def}{=} \{y \mid y \preceq x\}$ identifiziert, was eine Einbettung $P \hookrightarrow I_P$ liefert. Insbesondere gilt damit: $x \preceq y \iff I_x \subseteq I_y$. Außerdem lassen sich alle monotonen Funktionen (s. Abschnitt 10.2.4) $h : P \to C$ von P in irgendeine CPO C eindeutig zu stetigen Abbildungen $\hat{h} : I_P \to C$ erweitern.

Dieser Satz ist für uns rückblickend von großer Bedeutung, denn er zeigt, dass die lazy Listen aus Kapitel 2 tatsächlich eine CPO bilden und somit rekursive Gleichungen und Funktionen über diesen Listen wohldefiniert sind, also überhaupt einen Sinn haben. Denn auf Listen lässt sich sehr einfach die partielle Ordnung „a ist Anfang von b“ definieren:

$$a \preceq b \iff \exists r \bullet a +\!\!+ r = b$$

Dann gehört zu jeder Liste das Ideal aller ihrer Anfänge. Die Menge dieser Ideale ist nach Theorem 10.1 eine CPO und jede Kette von immer längeren Listen repräsentiert die zugehörige unendliche Liste.

Die Anwendung einer Funktion auf eine solche unendliche Liste darf aufgrund des Theorems dadurch berechnet werden, dass man die Funktion auf die endlichen Präfixe anwendet; denn das liefert eine Kette von Approximationen an das unendliche Resultat.

Mit anderen Worten: Was wir in Kapitel 2 eher intuitiv gemacht haben, gestützt auf Plausibilität und Anschauung, ist mathematisch wohlfundiert.

10.2.4 Fixpunkte

Wenn wir zu Funktionen $f(x) = \ldots\, x\, \ldots$ die Gleichungen der Art $x = f(x)$ betrachten, dann gibt es dazu oft viele Lösungen (die man auch als *Fixpunkte* bezeichnet), z. B.:

Funktion	*Gleichung*	*Fixpunkte*
$f(x) = 1^x$	$x = 1^x$	$\bot, 1$
$f(x) = \lfloor \sqrt[x]{27} \rfloor$	$x = \lfloor \sqrt[x]{27} \rfloor$	$\bot,\ 3$
$f(x) = x^2$	$x = x^2$	$\bot, 0, 1$
$f(S) = S \cup \{1, 7\}$	$S = S \cup \{1, 7\}$	$\bot,\ \{1, 7\},\ \{1, 7, 10\},\ \ldots$
$f(L) = 1 \cdot\!: L$	$L = 1 \cdot\!: L$	$\bot,\ \langle 1, 1, 1, \ldots \rangle$

Bei diesen Beispielen sind wir davon ausgegangen, dass die einzelnen Operationen *strikt* sind, weshalb $\bot$ immer zu den Lösungen gehört. (Gerade das will man in der Praxis in vielen Fällen verhindern, weshalb man andere Ordnungen zugrunde legt – z. B. eine nichtstrikte Ordnung für lazy Konstruktionen.) Eine Besonderheit zeigt die vierte Zeile; hier gibt es unendlich viele Fixpunkte, nämlich alle Mengen, die die Menge $\{1, 7\}$ umfassen.

Spannender werden die Fixpunkte, wenn wir rekursive Funktionen betrachten; denn diese sind ja nichts anderes als Gleichungen der Bauart $f = \text{«}\mathit{Rumpf}(f)\text{»}$. Beispielsweise sind für die Gleichung

$$\text{DEF } f = \lambda x, y \bullet \text{IF } x = y \text{ THEN } y + 1 \text{ ELSE } f(x, f(x - 1,\ y + 1)) \text{ FI}$$

folgende Funktionen Fixpunkte:

$$\begin{array}{llll} f_1 & \mathit{mit} \quad f_1(x, y) & = & \begin{cases} x + 1 & \mathit{if}\ x \geq y \\ y - 1 & \mathit{otherwise} \end{cases} \\ f_2 & \mathit{mit} \quad f_2(x, y) & = & x + 1 \\ f_{min} & \mathit{mit} \quad f_{min}(x, y) & = & \begin{cases} x + 1 & \mathit{if}\ x \geq y \wedge \mathit{even}(x - y) \\ \bot & \mathit{otherwise} \end{cases} \end{array}$$

Ein weiteres Beispiel ist

$$\text{DEF } g = \lambda x \bullet \text{IF } x = 0 \text{ THEN } 0 \text{ ELSE } g(x + 1) \text{ FI}$$

Fixpunkte gibt es hier sogar unendlich viele, darunter z. B.

$$g_i \quad mit \quad g_i(x) \quad = \begin{cases} 0 & if\ x = 0 \\ i & otherwise \end{cases}$$

$$g_{min} \quad mit \quad g_{min}(x) \quad = \begin{cases} 0 & if\ x = 0 \\ \bot & otherwise \end{cases}$$

Diese Beispiele deuten übrigens eine essenzielle Beobachtung an: Mehrere Fixpunkte kann es nur bei solchen Funktionen geben, bei denen f_{min} nicht terminiert (also $\bot$ ist).

Definition (Fixpunkt)

Ein Element x ist ein ***Fixpunkt*** der Funktion $f : A \to A$, wenn gilt $f(x) = x$.

Wir nennen $\bar{x}$ den ***kleinsten Fixpunkt***, wenn für alle anderen Fixpunkte y gilt: $\bar{x} \preceq y$.

Der ***bedingte kleinste Fixpunkt*** $\bar{x}_w$ ist definiert durch das kleinste $\bar{x}_w$ mit $w \preceq \bar{x}_w \wedge \bar{x}_w = f(\bar{x}_w)$; das heißt, $\bar{x}_w$ ist von allen Fixpunkten von f der kleinste, der w umfasst.

Kleinste Fixpunkte

Wenn man viele Fixpunkte hat, muss man versuchen, einen als „die“ Lösung auszuzeichnen. Hier kommen CPOs zum Tragen: Da in CPOs (monotone) Funktionen immer einen eindeutig bestimmten *kleinsten* Fixpunkt haben, kann man diesen jeweils als Standardlösung auszeichnen.

Die monotonen Funktionen garantieren zwar die Existenz eines kleinsten Fixpunktes, aber sie liefern kein konstruktives Verfahren zu seiner Bestimmung. Deshalb arbeitet man in der Praxis mit stetigen Funktionen.

Definition (monotone, stetige, inflationäre Funktionen)

Seien A und B CPOs. Dann gelten für Funktionen folgende Begriffe:

Eine Funktion $f : A \to B$ ist ***monoton***, wenn sie die Ordnung der Argumente erhält: $x \preceq y \implies f(x) \preceq f(y)$.

Eine Funktion $f : A \to B$ heißt ***stetig***, wenn sie die kleinsten oberen Schranken erhält: $f(\sqcup K) = \sqcup(f * K)$. Jede stetige Funktion ist insbesondere auch monoton.

Eine Funktion $f : A \to A$ heißt ***inflationär***, wenn gilt: $x \preceq f(x)$.

Der für unsere Zwecke entscheidende Satz ist der klassische Fixpunktsatz, der auf Tarski (1942) und Kleene (1952) zurückgeht:

Theorem 10.2 (Fixpunktsatz). *Jede stetige Funktion f auf einer CPO hat einen kleinsten Fixpunkt $\bar{x}$; dieser Fixpunkt ergibt sich als die kleinste obere Schranke der so genannten Kleene-Kette $\bar{x} = \sqcup\{\bot, f(\bot), f^2(\bot), f^3(\bot), \dots\}$.*

Beweis. Der Beweis ist relativ einfach; er besteht aus zwei Teilen:

a) Das Supremum der Kleene-Kette ist ein Fixpunkt: Denn wegen der Stetigkeit von f gilt:
$f(\sqcup\{f^i(\bot)\}) = \sqcup(f * \{f^i(\bot)\}) = \sqcup\{f^{i+1}(\bot)\} = \sqcup\{f^i(\bot)\}$.

b) Dieses Supremum ist kleiner als jeder Fixpunkt y, also $\sqcup\{f^i(\bot)\} \preceq y$:
Induktionsanfang: $\bot \preceq y$.
Wegen der Monotonie von f gilt dann: $f^i(\bot) \preceq y \implies f(f^i(\bot)) \preceq f(y) = y$.
Damit sind alle Elemente der Kleene-Kette kleiner als y, woraus folgt, dass auch das Supremum (definitionsgemäß) kleiner als y ist.

Anmerkung: Natürlich lässt sich diese ganze Begrifflichkeit auch dual auf größte *Fixpunkte übertragen. In der Programmierung lässt sich das aber ganz einfach dadurch realisieren, dass wir die entsprechenden Strukturen und Theorien mit „ $\geq$ “ anstelle von „ $\leq$ “ instanziieren.*

Endliche Fixpunkte

Für unsere Zwecke ist diese allgemeine Form des Fixpunktsatzes nicht immer nützlich, da die Kleene-Kette potenziell unendlich ist und ihre endlichen Abschnitte den Fixpunkt deshalb im Allgemeinen nur approximieren. Wir sind hier aber an terminierenden Programmen interessiert und müssen deshalb restriktivere Situationen betrachten. Offensichtlich gilt:

Korollar 10.1 (Endliche Fixpunkte).

1. *Wenn in einer Kleene-Kette zwei aufeinanderfolgende Elemente* gleich *sind, also $f^i(\bot) = f^{i+1}(\bot)$, dann ist $\bar{x} \stackrel{def}{=} f^i(\bot)$ der kleinste Fixpunkt (weil von da an auch alle folgenden Elemente gleich sind). In diesem Fall reicht übrigens Monotonie von f bereits aus.*
2. *Zur Berechnung des* bedingten kleinsten Fixpunktes $\bar{x}_w$ *lässt man die Kleene-Kette nicht mit $\bot$, sondern mit w beginnen; dann erhält man $\bar{x}_w = \sqcup\{w, f(w), f^2(w), f^3(w), \dots\}$. Voraussetzung ist allerdings, dass f inflationär für w ist: $w \preceq f(w)$.*

Beweis. Die erste der beiden Beobachtungen ist trivial. Auch die zweite ist leicht einzusehen: Die Teilmenge $A' = \{x \in A \mid w \preceq x\}$ ist eine CPO mit w als kleinstem Element. Da f als inflationär vorausgesetzt wird, ist f eine Abbildung von A' in A'. (In [28] findet sich ein Beispiel dafür, dass diese Bedingung wirklich notwendig ist.)

Es gibt eine Reihe von offensichtlichen Kriterien, die die geforderte Endlichkeit garantieren, zum Beispiel:

- Definitions- oder Wertebereich von f sind kettenendlich.
- Der kleinste Fixpunkt von f ist endlich.
- f ist von der (in der Praxis häufig anzutreffenden) Form $f(x) = x \sqcup g(x)$, wobei g monoton ist und einen kettenendlichen Wertebereich hat.

Optimierung durch „Mikroschritte“

Eine wesentliche Erkenntnis zur Optimierung der Berechnung ist, dass wir häufig in granulareren Schritten vorgehen können. Ein typisches Beispiel dafür ist die iterative Lösung von Gleichungssystemen in der Numerischen Mathematik. Eine Methode ist z. B. das Gauß-Jordan-Verfahren, das in Hauptschritten vorgeht, bei denen jeweils die neue Matrix $M^{(i+1)}$ komplett aus der alten Matrix $M^{(i)}$ abgeleitet wird. Demgegenüber stellt das Gauß-Seidel-Verfahren eine deutliche Beschleunigung dar: Der Trick ist, dass für die Berechnung jedes Wertes von $M^{(i+1)}$ schon möglichst viele vorhandene Werte aus der neuen Matrix $M^{(i+1)}$ selbst genommen werden und nicht die alten aus $M^{(i)}$.

Analog können wir bei Gleichungssystemen vorgehen. Wenn wir für eine Variable x_1 ihren neuen Approximationswert berechnet haben, dann sollten wir für die Berechnung von x_2 nicht den alten Wert von x_1 hernehmen, sondern schon den neuen Wert von x_1. Das heißt, wir erhalten zum Beispiel die Berechnung

$$x_2^{(i+1)} = f_2(x_1^{(i+1)}, x_2^{(i)}, \ldots, x_n^{(i)})$$

Ähnlich ist es auch bei den mengenbasierten Fixpunktberechnungen. Schauen wir uns noch einmal das Beispiel der Erreichbarkeit in Graphen an.

$$\begin{aligned} \text{DEF } \mathit{reachable}(G)(x) = \; & \mathit{fixpoint}(\mathit{succs})(\{x\}) \\ & \text{WHERE } \mathit{succs}(S) = S \cup \textstyle\bigcup_{x \in S} G.\mathit{succ}(x) \end{aligned}$$

Die entscheidende Berechnung ist jeweils die Auswertung der lokalen Funktion $\mathit{succs}(S) = S \cup \bigcup_{x \in S} G.\mathit{succ}(x)$. Das Problem ist, dass wir von einem Iterationsschritt zum nächsten für alle „alten“ Knoten die Menge $\mathit{succs}(\ldots)$ wieder und wieder ausrechnen. Dabei reicht es doch, wenn wir immer nur für die neu hinzugekommenen Knoten die Funktion *succs* berechnen.

Das in diesen Beispielen skizzierte Optimierungsprinzip wollen wir jetzt in unserem Kontext realisieren. Erinnern wir uns: Die (verallgemeinerte) Kleene-Folge hat die Form

$$\{w, f(w), f^2(w), f^3(w), \ldots, f^i(w), f^{i+1}(w), \ldots\}$$

In der Praxis kann man die *„Hauptschritte“* $f^i(w) \mapsto f^{i+1}(w)$ oft in *„Mikroschritte“* zerlegen:

$$\{\ldots, f^i(w), s_{i_1}, s_{i_2}, \ldots, s_{i_n}, f^{i+1}(w), \ldots\}$$

Die Verallgemeinerung geht sogar noch ein Stück weiter: Wie man zum Beispiel bei der Lösung von Gleichungssystemen sieht, laufen die Mikroschritte oft nicht einmal synchron zu den Hauptschritten. Damit betrachten wir also (nahezu beliebige) Folgen der Bauart

$$\{w = s_0, s_1, s_2, \ldots, s_i, s_{i+1}, \ldots\}$$

Das folgende Theorem beantwortet die Frage, wann eine solche Folge auf einen Fixpunkt führt.

Theorem 10.3 (Verallgemeinerte Fixpunkt-Kette). *Wenn für eine Folge* $\{w = s_0, s_1, s_2, \ldots\}$ *und eine monotone Funktion* f *gilt:*

$$s_i \prec s_{i+1} \preceq f(s_i) \quad \vee \quad s_i = s_{i+1} = f(s_i)$$

dann hat sie folgende Eigenschaften (siehe [28]):

1. *Wenn* $s_k = f(s_k)$*, dann gilt* $s_k = \bar{x}_w$ *(und die Folge endet).*
2. *Wenn* $\bar{x}_w$ *endlich berechenbar ist, dann gibt es auch ein Element* s_k *in der Folge mit* $s_k = \bar{x}_w$*.*

Beweis.

1. Durch Induktion über i sieht man sofort, dass jedes Element s_i von einem $f^j(w)$ dominiert ist, d. h. $\forall i\ \exists j : s_i \preceq f^j(w)$. Daraus folgt: $s_k \preceq \bar{x}_w$ und somit auch $s_k = \bar{x}_w$.
2. $\exists j : f^j(w) = f^{j+1}(w)$. Daraus folgt wegen (1): $\forall i : s_i \preceq f^j(w)$. Daraus folgt wiederum: Es gibt nur endlich viele $s_i \prec f^j(w)$, da die Endlichkeit des Fixpunkts keine unendlichen Ketten zulässt.

Man beachte, dass das Theorem für *unendliche* Fixpunkte nur sehr schwache Aussagen liefert. Denn es ist nicht ausgeschlossen, dass es im Bereich $s_i, \ldots, f(s_i)$ unendlich viele Mikroschritte s_j gibt, so dass der Algorithmus nie zum Fixpunkt konvergiert, sondern sich zwischen zwei sehr frühen Approximationen verliert.

Wichtig ist das Theorem aber für den praktisch relevanten Fall der Berechnung *endlicher* Fixpunkte. Dabei ist für die spätere algorithmische Behandlung von Gleichungssystemen folgender Spezialfall nützlich, der gerade die oben erwähnte Optimierung des Gauß-Seidel-Verfahrens widerspiegelt:

Korollar 10.2. *Wenn* f *die spezielle Form hat*

$$f(x) = f_1(x) \sqcup \ldots \sqcup f_n(x)$$

dann können wir die s_i *jeweils aus einem der speziellen „Worksets" nehmen, bei denen wir jeweils nur eine der Funktionen* f_j *benutzen:*

$$\Delta_j(x, f) = \{ s \mid x \prec s \preceq f_j(x) \}$$

Beweis. Es gilt: $x \prec s \preceq f_j(x) \preceq f_1(x) \sqcup \ldots \sqcup f_n(x) = f(x)$.

Man beachte, dass dies bei der Wahl $f_1 = id$ und $f_2 = g$ den schon früher erwähnten Spezialfall

$$f(x) = x \sqcup g(x)$$

einschließt. Das ist genau die Form unseres *Reachability*-Beispiels in Programm 10.1 (nach den Adaptionen).

Damit haben wir die Rahmenbedingungen fixiert, unter denen wir nach Algorithmen suchen können, die den Fixpunkt in geeigneten Mikroschritten berechnen.

10.3 Die Programmierung von Fixpunkt-Algorithmen

Der Strebende ist Gottes Freund.
Scheich Dschelal Eddin Rumi

Nach den mathematischen Vorübungen zur Klärung der theoretischen Grundlagen müssen wir jetzt die Idee der Fixpunktberechnung programmiersprachlich umsetzen. Dazu sind drei Dinge zu tun:

- Das Konzept der CPOs muss programmiersprachlich erfasst werden; dazu dient das Mittel der Typklassen.
- Der Fixpunktalgorithmus muss programmiert werden; das geschieht in einer entsprechenden Struktur.
- Man muss die Datentypen identifizieren, die CPOs sind, weil nur für diese der Algorithmus funktioniert.

Diese Fragen werden in den folgenden Abschnitten der Reihe nach diskutiert.

10.3.1 CPOs als Typklasse

Das Prinzip der Fixpunktiteration ist nur anwendbar, wenn die zugrunde liegenden Daten eine CPO bilden. Deshalb müssen wir als Erstes das Konzept der CPOs im Rahmen unserer Programmiersprache ausdrücken. Dazu dient das Mittel der *Typklassen*.

In Programm 10.3 stellen wir eine Spezifikation *CompletePartialOrder* mit Typklasse *Cpo* bereit. Dabei verwenden wir die bereits in Abschnitt 9.6 im Programm 9.8 definierte Spezifikation *PartialOrder* und erweitern sie um ein Bottom-Element und die Operation $\sqcup$ (*least upper bound*). Außerdem gelten für diese Operatoren die Monoidgesetze, sowie Kommutativität etc. (Wir verwenden die traditionelle Signatur-Notation.)

Programm 10.3 Die Spezifikation *CPO*

```
SPECIFICATION CompletePartialOrder = {
  EXTEND PartialOrder, Monoid RENAMING (∘, unit) AS (⊔, ⊥)
  TYPE α: Cpo
  FUN ⊥: α                                          -- Bottom
  FUN _ ⊔ _: α × α → α                              -- kleinste obere Schranke
  PROP ∀a • ⊥ ⪯ a                                   -- kleinstes Element
  PROP ∀a, b •  a ⪯ a ⊔ b  ∧  b ⪯ a ⊔ b             -- obere Schranke
  PROP ∀a, b, c • a ⪯ c ∧ b ⪯ c ⟹ a ⊔ b ⪯ c         -- kleinste obere Schranke
       ...
}
TYPECLASS Cpo = CompletePartialOrder.α
```

10.3.2 Fixpunktberechnung: Der Basisalgorithmus

Der Fixpunktsatz liefert sofort einen simplen Algorithmus zur Berechnung des Fixpunkts – sofern die Kleene-Kette endlich ist. Aus Gründen der Lesbarkeit spezifizieren wir den (bedingten) kleinsten Fixpunkt (in Anlehnung an die Notation in [28]) als

$$\begin{array}{ll} \text{LEAST } x \bullet x = f(x) & -- \textit{ kleinster Fixpunkt von } f \\ \text{LEAST } x \bullet w \preceq x \wedge x = f(x) & -- \textit{ bedingter kleinster Fixpunkt von } f \end{array}$$

Wir teilen die Definition des Fixpunktalgorithmus in zwei Teile auf. In Programm 10.4 wird zunächst die Aufgabenstellung *spezifiziert*. Dabei sehen wir zwei Funktionen vor, eine für den Fixpunkt und eine für den bedingten Fixpunkt. Mit Hilfe von Overloading kann der gleiche Name für beide Funktionen benutzt werden.

Außerdem geben wir noch eine Variante an, die die Kleene-Kette selbst als *lazy Liste* generiert; dazu verwenden wir einen Typ *List*, wie er in Abschnitt 2.3 eingeführt wurde. Diese Variante kann dann in Programmen benutzt werden, die bei nicht-endlichen Fixpunkten wenigstens eine geeignete Approximation suchen.

Programm 10.4 Die Spezifikation *FixpointSpec*

```
SPECIFICATION FixpointSpec = {
  FUN fixpoint: [α: Cpo] ↦ (α → α) → α                    -- kleinster Fixpunkt
  PROP fixpoint(f) = LEAST x • x = f(x) ∧ finite(x)
  FUN fixpoint: [α: Cpo] ↦ (α → α) → α → α                -- bedingter kl. Fixpunkt
  PROP fixpoint(f)(w) = LEAST x • w ⪯ x ∧ x = f(x) ∧ finite(x)
  FUN kleene: [α: Cpo] ↦ (α → α) → List α                 -- Kleene-Kette
  PROP kleene(f) = ⟨⊥, f(⊥), f²(⊥), ... ⟩
  FUN kleene: [α: Cpo] ↦ (α → α) → α → List α             -- bed. Kleene-Kette
  PROP kleene(f)(w) = ⟨w, f(w), f²(w), ... ⟩
}
```

Man beachte, dass wir hier ein bisschen mogeln: Das Prädikat *finite*(x) soll ausdrücken, dass der Fixpunkt x endlich ist (weil sonst der Algorithmus nicht terminiert). Aber dieses Prädikat ist nur informell durch seinen Namen erklärt; wir haben es nirgends definiert.[4]

In Programm 10.5 werden diese Funktionen *konstruktiv definiert*. Das geschieht im Rahmen einer Struktur *Fixpoint*, die die Spezifikation realisiert (was durch die entsprechende Typisierung ausgedrückt wird).

[4] Eine formale Definition für das Prädikat *finite* führt auf grundlegende theoretische Probleme; sie würde aber praktisch auch nicht viel helfen. Denn was man wirklich braucht, ist eine Aufwandsabschätzung.

Programm 10.5 Die Struktur *Fixpoint*

```
STRUCTURE Fixpoint: FixpointSpec = {
  DEF fixpoint(f) = fixpoint(f)(⊥)
  DEF fixpoint(f)(w) = IF f(w) = w THEN w ELSE fixpoint(f)(f(w)) FI
  DEF kleene(f) = kleene(f)(⊥)
  DEF kleene(f)(w) =  chain WHERE chain = w ·: (f ∗ chain)
}
```

Die Implementierung ist eine unmittelbare Umsetzung des Fixpunktsatzes. Man beachte, dass die Eigenschaft α: *Cpo* insbesondere auch α: *Eq* nach sich zieht, der Vergleich $f(x) = x$ also syntaktisch möglich ist.

Die Terminierung der Funktion *fixpoint* hängt davon ab, ob die Eigenschaft *finite*(x) der Spezifikation tatsächlich erfüllt ist.

10.3.3 Optimierung durch feinere Granularität

Die Algorithmen in Programm 10.5 setzen das Prinzip der Kleene-Folge unmittelbar in Code um. Das ist zwar garantiert korrekt (unter Beachtung der notwendigen Randbedingungen), aber im Allgemeinen nicht effizient. Deshalb hatten wir in Abschnitt 10.2.4 ein Optimierungsprinzip studiert, das auf so genannten Mikroschritten basiert. Dass diese Optimierungen korrekt sind, folgt aus dem Theorem 10.3.

Wie wir in Abschnitt 10.2.4 gesehen haben, basiert der Algorithmus ganz zentral auf dem Konzept der Worksets, die in Korollar 10.2 verwendet werden.

$$\Delta_j(x, f) = \{s \mid x \prec s \preceq f_j(x)\}$$

Wir wollen diese Worksets allerdings noch etwas näher an die tatsächlichen Gegebenheiten der praktischen Anwendungen heranbringen. Dort haben wir es typischerweise mit Inkrementen zu tun, das sind z. B. die Elemente, die neu zu einer Menge hinzukommen oder die Variablen, deren Wert verbessert wurde. Das spiegelt sich in folgender Variation wider:

$$\Delta_j(x, f) = \{d \mid x \prec x \oplus d \preceq f_j(x)\}$$

Die d haben hier die Rolle der Inkremente, und der Operator $x \oplus d$ liefert die Vergrößerung von x um das Inkrement d.

Damit wir die entsprechende Variante der Fixpunktbildung programmieren können, müssen wir als Voraussetzung die Generierung solcher Worksets von Inkrementen verfügbar haben. Mit anderen Worten, der Fixpunktalgorithmus muss auf Typen α aufbauen, die nicht nur CPOs sind, sondern auch noch Worksets besitzen. Das ist in Programm 10.6 spezifiziert.

Um auch diese Idee in ein konkretes Programm umzusetzen, betrachten wir das verallgemeinerte Schema zur Fixpunktberechnung in Mikroschritten,

Programm 10.6 Die Spezifikation *Incremental*

```
SPECIFICATION Incremental = {
  EXTEND CompletePartialOrder
  TYPE α: Incremental
  FUN Δ: α × α → Set β
  FUN _ ⊕ _: α × β → α
  PROP Δ(s, fs) = {d | s ≺ s ⊕ d ⪯ fs}
  PROP Δ(s, fs) = ∅ ⟺ fs ⪯ s
}
TYPECLASS Incremental = Incremental.α
```

das in der Struktur *Fixpoint2* in Programm 10.7 definiert ist. Dabei sehen wir wieder die beiden Varianten der Fixpunktbildung vor. Man beachte, dass auch diese Struktur die Spezifikation *FixpointSpec* erfüllt, weil die Typklasse *Incremental* eine Spezialisierung von *Cpo* ist.

Programm 10.7 Fixpunktberechnung in Mikroschritten

```
STRUCTURE Fixpoint2: FixpointSpec = {
  FUN fixpoint: [α: Incremental] ↦ (α → α) → α          -- kleinster Fixpunkt
  DEF fixpoint(f) = fixpoint(f)(⊥)
  FUN fixpoint: [α: Incremental] ↦ (α → α) → α → α      -- bed. kl. Fixpunkt
  DEF fixpoint(f)(w) = iterate(f)(w, Δ(w, f(w)))
  FUN iterate: (α → α) → α × Set β → α
  DEF iterate(f)(s, workset) =
      IF workset = ∅ THEN s
                     ELSE LET d = arb(workset)
                              s' = s ⊕ d
                          IN
                              iterate(f)(s', Δ(s', f(s'))) FI
  FUN kleene: [α: Incremental] ↦ (α → α) → List α       -- Kleene-Kette
  DEF kleene(f) = kleene(f)(⊥)
  FUN kleene: [α: Incremental] ↦ (α → α) → α → List α
  DEF kleene(f)(w) = chain WHERE chain = w ·: (h * chain)
                                 h(s) = s ⊕ arb(Δ(s, f(s)))
}
```

Damit wird es allerdings wichtig, dass die Funktion Δ „billig“ ist. Dafür gibt es mehrere Möglichkeiten:

- Es kann sein, dass die Menge $M = \Delta(s, f(s))$ nur „virtuell“ auftritt. Das heißt, wir haben ein effizientes Prädikat, das direkt aussagt, ob die Menge M leer wäre, und eine Funktion, die uns direkt $d = arb(M)$ liefert.
- Ein häufig funktionierender Trick ist das so genannte *Finite differencing*. Hier berechnet man den neuen Wert des Parameters *workset* nicht (wie in Programm 10.7) durch den Aufruf von Δ, sondern durch eine „billigere“ Funktion Δ', für die gilt $\Delta'(s, d, \Delta(s, f(s))) = \Delta(s \oplus d, f(s \oplus d))$.

Wir illustrieren diese abstrakten Entwicklungen an unserem einleitenden Beispiel der Erreichbarkeit in Graphen.

Beispiel 10.1 (Nochmals: Erreichbarkeit in Graphen)

Wir betrachten nochmals das Problem der Erreichbarkeit in Graphen. Die grundlegende Funktion, über der die Fixpunktbildung stattfindet, ist

$$succs(S) = S \cup \bigcup_{x \in S} G.succ(x)$$

die die Nachbarn aller Knoten in S bestimmt.

Wir wählen die beiden Funktionen $\Delta(S, f(S)) = f(S) - S$ (also nur die jeweils neu hinzugekommenen Elemente) und $S \oplus d = S \cup \{d\}$. Damit sieht unser Programm folgendermaßen aus (wobei wir die Parameterbezeichnungen so instanziieren, dass sie die Lesbarkeit erleichtern):

```
DEF reachable(G)(x) = fixpoint(G.succ)(∅, {x})
DEF fixpoint(succs)(s, workset) =
    IF workset = ∅ THEN s
                   ELSE LET d = arb(workset)
                            s' = s ∪ {d}
                            w' = workset ∪ succs(d) − s'
                        IN
                            fixpoint(succs)(s', w')
                   FI
```

Das entspricht $\Delta'(s, d, workset) = workset \cup G.succ(d) - (s \cup \{d\})$.

Wenn man die Menge s als „schwarz“ und *workset* als „grau“ bezeichnet, hat man übrigens genau den Algorithmus, der Knoten sukzessive von weiß (noch nicht betrachtet) über grau (steht zur Bearbeitung an) in schwarz (fertig) umfärbt.

10.3.4 Verallgemeinerungen und Variationen

Die im Software-Engineering oft propagierte Idee der *Design Patterns* umfasst unter anderem, dass man auch Variationen des Themas betrachtet, die sich auf die angesprochene Lösung zurückführen oder zumindest in analoger Weise behandeln lassen. In [28] werden eine Reihe solcher Adaptionen diskutiert, auf die wir hier kurz eingehen wollen.

Unter sehr schwachen Annahmen (siehe [28]) kann man verschiedene Varianten der Fixpunktbildung ineinander überführen. Das heißt, die folgenden drei Fixpunkte stimmen meistens überein:

$$\begin{aligned}
&\text{LEAST } x \bullet x = w \sqcup f(x) &&= \mathit{fixpoint}(\lambda x \bullet w \sqcup f(x)) \\
&\text{LEAST } x \bullet w \preceq x \wedge x = x \sqcup f(x) &&= \mathit{fixpoint}(\lambda x \bullet x \sqcup f(x))(w) \\
&\text{LEAST } x \bullet w \preceq x \wedge x = f(x) &&= \mathit{fixpoint}(f)(w)
\end{aligned}$$

Hilfreich ist es oft auch, die beteiligten Prädikate umrechnen zu können. Die wichtigste dieser Umformungen ist wohl der bekannte Elementarzusammenhang zwischen Supremum und partieller Ordnung in CPOs (Verbänden):

$$(f(x) \preceq x) \iff (x = x \sqcup f(x))$$

Doch es gibt weitere Umformungen, die oft helfen können, aus entsprechenden Spezifikationen gute Algorithmen abzuleiten:

$$\begin{aligned}
&(\text{LEAST } x \bullet x = f(x) \wedge x = g(x)) \\
&\qquad = \text{LEAST } x \bullet x = (\mathit{fixpoint}(f) \sqcup \mathit{fixpoint}(g)) \\
&\qquad = \mathit{fixpoint}(\mathit{fixpoint}(f) \circ \mathit{fixpoint}(g)) \\
&\qquad = \mathit{fixpoint}(\mathit{fixpoint}(g) \circ \mathit{fixpoint}(f)) \\
&(\text{LEAST } x \bullet x = f(x) \vee x = g(x)) = \mathit{min}(\mathit{fixpoint}(f), \mathit{fixpoint}(g)) \\
&(\text{LEAST } x, y \bullet x = f(y) \wedge y = g(x)) = \text{LEAST } x \bullet x = f(g(x)) \\
&\qquad = \mathit{fixpoint}(f \circ g) \\
&(\text{LEAST } x \bullet u \preceq x \wedge v \preceq x \wedge x = f(x)) = \mathit{fixpoint}(f)(u \sqcup v)
\end{aligned}$$

10.3.5 Fixpunkte als *„Design Pattern“*

Zu Beginn dieses Kapitels hatten wir – als Einstieg und zur Motivation – eine einfache Lösung des Erreichbarkeitsproblems in Graphen formuliert. Dieser Ansatz lässt sich auf alle möglichen Arten von transitiven Hüllen übertragen, so dass das obige Beispiel als Phänotyp („*Design Pattern*“) für eine ganze Klasse von Problemen gesehen werden kann. Dabei sind jeweils folgende Dinge zu tun:

1. Bringe die Aufgabenstellung in die Form eines „Minimumproblems“, also in eine Spezifikation der Bauart $\text{LEAST } x \bullet w \preceq x \wedge p(x)$.
2. Wandle das Prädikat p so um, dass ein Fixpunktproblem entsteht, die Spezifikation also lautet: $\text{LEAST } x \bullet w \preceq x \wedge x = f(x)$.
3. Prüfe, ob die Anwendbarkeitsbedingungen erfüllt sind (Monotonie, Endlichkeit, ...).
4. Instanziiere die Struktur *Fixpoint* und wende die Funktion *fixpoint* an.

10.4 Datentypen als CPOs

In der Programmierung treffen wir auf viele Strukturen, die CPOs darstellen.[5] Im Folgenden wollen wir einige dieser Strukturen auflisten. Dabei machen wir uns das Sprachmittel der Spezifikationen und Typklassen zunutze.

Boolesche Werte

Die Struktur *Bool* liefert auf zwei duale Weisen eine CPO. Dies wird in Programm 10.8 explizit gemacht.

Programm 10.8 Die Struktur *Boolean* als CPO

```
STRUCTURE Boolean = {
  TYPE Bool = { false, true }
  FUN _ ∧ _: Bool × Bool → Bool
  DEF (a ∧ b) = ...
  ...
}
PROP Boolean: CompletePartialOrder
        RENAMING (α, ⊥, ⊑, ⊔)AS (Bool, false, ⟹ , ∨)
PROP Boolean: CompletePartialOrder
        RENAMING (α, ⊥, ⊑, ⊔)AS (Bool, true, ⟸, ∧)
```

Wie wir im vorigen Abschnitt gesehen haben, ist es auch relevant, ob eine gegebene Funktion monoton und inflationär ist. Für selbstprogrammierte Funktionen ist dies natürlich jeweils zu beweisen. Für einige wichtige vordefinierte Funktionen ist das in Tabelle 10.1 enthalten. Diese Tabelle bezieht sich auf die erste CPO-Instanz mit *false* und $\vee$.

Operation	monoton in ...	inflationär in ...
$p \implies q$	q	q
$p \vee q$	p, q	p, q
$p \wedge q$	p, q	

Tab. 10.1: Eigenschaften wichtiger Boolescher Operationen

[5] Dies darf nicht damit verwechselt werden, dass – wie in Abschnitt 10.2.2 erwähnt – die ganze Semantiktheorie von Programmiersprachen ebenfalls auf CPOs basiert. Die Ordnung dieser CPOs spiegelt den „Informationszuwachs“ von Inkarnation zu Inkarnation einer rekursiven Prozedur oder einer Schleife wider. Sie ist also am Berechnungsablauf von Programmen orientiert, nicht an Relationen zwischen Daten.

Mengen

Aus der Mathematik ist wohl bekannt, dass die Teilmengen einer gegebenen Menge M mit der Relation $\subseteq$ eine CPO bilden, deren Bottom-Element die leere Menge $\emptyset$ ist. Die Rolle der *Join*-Operation $\sqcup$ übernimmt die Mengenvereinigung $\cup$. Das überträgt sich natürlich auf unsere Datenstruktur *Set* α, wie in Programm 10.9 angegeben.

Programm 10.9 Die Struktur *Set* als CPO

```
STRUCTURE Set = {
  TYPE Set α
  FUN ∅: Set α                                  -- leere Menge
  FUN {_}: α → Set α                            -- einelementige Menge
  FUN arb _: Set α → α                          -- Auswahl eines bel. Elements
  FUN _ ∪ _: Set α × Set α → Set α              -- Vereinigung
  FUN _ ⊆ _: Set α × Set α → Bool               -- Teilmengen-Relation
  FUN _ ∈ _: α → Set α → Bool                   -- Element-Beziehung
  FUN |_|: Set α → Nat                          -- Kardinalität
  FUN _ ∩ _: Set α × Set α → Set α              -- Schnittmenge
  FUN _ − _: Set α × Set α → Set α              -- Differenzmenge
  FUN _ × _: Set α × Set α → Set (α × α)        -- Mengenprodukt
  FUN _ ▷ _: Set α × (α → Bool) → Set α         -- Mengenfilter
  FUN _ has _: Set α × (α → Bool) → Bool        -- Existenztest
  FUN _ ensures _: Set α × (α → Bool) → Bool    -- Gültigkeitstest
      ...
}
PROP Set: CompletePartialOrder RENAMING (α, ⊥, ≼, ⊔) AS (Set α, ∅, ⊆, ∪)
```

Auch hier erhält man eine duale CPO, die auf dem Durchschnitt $\cap$ anstelle der Vereinigung aufbaut. Allerdings geht das nur, wenn der Typ α der Mengenelemente *endlich* ist; dann übernimmt die Menge M aller Elemente vom Typ α die Rolle von $\bot$.

Für Standardoperationen auf Mengen haben wir wichtige Monotonie-Eigenschaften in Tabelle 10.2 zusammengefasst. Dabei verwenden wir für die Operationen $\triangleright$, *has* und *ensures* die geläufigeren mathematischen Notationen.

Anmerkung: Einen wichtigen Aspekt haben wir in Programm 10.9 weggelassen: Mengen lassen sich nicht über beliebigen Elementtypen α definieren. Die Operation $x \in s$ ist nur implementierbar, wenn auf α eine Gleichheit existiert. Und die Operation arb(s) setzt sogar voraus, dass α geordnet ist; denn nur so lässt sich garantieren, dass arb in der Tat eine Funktion ist und nicht nur eine Relation. Mit anderen Worten, wir müssen die Einschränkung α: Ord fordern. Da diese Ordnung aber nichts mit der hier diskutierten CPO-Eigenschaft von Set zu tun hat, haben wir diesen Aspekt weggelassen, um nicht unnötige Verwirrung zu stiften.

Operation		monoton in ...	inflationär in ...
$x \in S$		S	
$\|S\|$		S	
$S \cap T$		S, T	
$S \cup T$		S, T	S, T
$S - T$		S	
$S \times T$		S, T	
$S \triangleright p$	$\{x \in S \mid p(x)\}$	S, p	
S *has* p	$\exists x \in S \bullet p(x)$	S, p	
S *ensures* p	$\forall x \in S \bullet p(x)$	p	

Tab. 10.2: Eigenschaften wichtiger Mengenoperationen

Sequenzen

Sequenzen lassen sich auf sehr leicht als CPO auffassen, wenn man sich mit einer stark eingeschränkten Sichtweise zufrieden gibt. Diese reicht aber aus, um den wichtigsten Anwendungsfall zu behandeln, nämlich die Definition von unendlichen Listen als Suprema von aufsteigenden Ketten von Sequenzen. Diese Sicht haben wir in Programm 10.10 angegeben.

Programm 10.10 Die Struktur *Sequence* als CPO

```
STRUCTURE Sequence = {
  TYPE Seq α
  FUN ◊: Seq α
  FUN _ ⊑ _: Seq α × Seq α → Bool          -- "ist-Anfang-von"-Relation
  FUN _ ⊔ _: Seq α × Seq α → Seq α         -- kombinieren
  FUN _ ∈ _: α × Seq α → Bool              -- Element-Beziehung
  FUN _ ++ _: Seq α × Seq α → Seq α        -- Konkatenation
  FUN rest^ _ _: Nat × Seq α → Seq α       -- Streichen der ersten i Elemente
  DEF rest^i S = ...
      ...
}
PROP Sequence: CompletePartialOrder RENAMING (α, ⊥, ≺, ⊔) AS (Seq α, ◊, ⊑, ⊔)
```

Die Operation $(a \sqcup b)$ ist nur definiert, wenn eine der beiden Sequenzen Anfang der anderen ist, also $(a \sqsubseteq b) \vee (b \sqsubseteq a)$ gilt. Das Ergebnis ist dann die längere der beiden Sequenzen.

Für den Fall, dass der Elementtyp α selbst schon eine CPO ist, kann man diese Definition etwas verallgemeinern: Dann braucht für $a \sqsubseteq b$ nicht unbedingt a Anfang von b zu sein, sondern es genügt, dass die Elemente von a jeweils kleiner sind als die entsprechenden Elemente von b. Darauf gehen wir gleich im Zusammenhang mit Maps gleich noch genauer ein.

Die Monotonie-Eigenschaften wichtiger Sequenzoperationen sind in Tabelle 10.3 aufgelistet.

Operation	monoton in ...	inflationär in ...
$x \in s$	s	
$s \sqsubseteq t$	t	
$s +\!\!+ t$	t	s
$rest^i(s)$	s	

Tab. 10.3: Eigenschaften wichtiger Sequenzoperationen

Anmerkung: Es ist erstaunlich kompliziert, eine weniger spezielle CPO-Sicht von Sequenzen zu geben. Man könnte z. B. versuchen, eine allgemeinere Teilsequenz-Relation $a \subseteq b \iff \exists u, v \bullet b = u +\!\!+ a +\!\!+ v$ zu definieren; das heißt, a tritt kompakt irgendwo in b auf. (Es reicht nicht, wenn die Elemente von a nur verstreut in b vorkommen.)

Die Operation $a \sqcup b$ muss dann aber sehr aufwendig definiert werden, um tatsächlich die kleinste obere Schranke von a und b zu bestimmen. Dies läuft im Wesentlichen auf folgende Situation hinaus: $(a' +\!\!+ u) \sqcup (u +\!\!+ b') = (a' +\!\!+ u +\!\!+ b')$. Mit anderen Worten, $\sqcup$ ist definiert, wenn die beiden Argumentsequenzen teilweise überlappen. Das gilt natürlich auch für die duale Situation, bei der der Anfang von b mit dem Ende von a übereinstimmt. Falls beides gilt, ist die Funktion wegen der resultierenden Mehrdeutigkeit undefiniert.

Diese Variante ist aber nicht nur sehr komplex, sondern verhindert auch weitgehend die Monotonieeigenschaften der anderen Operationen.

Maps und Arrays

In der Funktionalen Programmierung sind Abbildungen generell wichtig. Ein wichtiger Spezialfall sind *endliche Abbildungen*, also Abbildungen mit *endlichem Definitionsbereich*; anstelle von Definitionsbereich spricht man dann auch oft von der *Indexmenge*. Die häufigste Implementierungstechnik für diese Maps sind die so genannten Hash-Tabellen, die auch in Sprachen wie JAVA zum Standardrepertoire gehören. Ein Spezialfall der Maps sind die Arrays, bei denen der Indexbereich ein Intervall der ganzen Zahlen ist. Genauer wird das in den Kapiteln 14 und 15 betrachtet werden, wo insbesondere gezeigt wird, dass sowohl Maps als auch Arrays nur Spezialfälle von Funktionen sind.

Auch über Maps und Arrays lassen sich CPOs aufbauen, wobei man im Wesentlichen zwei verschiedene Möglichkeiten (und ihre Kombination) hat:

1. Man kann Maps und Arrays über beliebigen Elementtypen zu CPOs machen, indem man inkrementell wachsende Indexbereiche nimmt.
2. Man kann auch Maps und Arrays mit festen Indexmengen zu CPOs machen, sofern der Elementtyp eine CPO ist.

Anmerkung 1: Wir zeigen hier den Fall der Mappings aus Abschnitt 15.3 (Seite 327). Die Prinzipien gelten aber auch für den allgemeineren Fall der Functions aus Abschnitt 15.2 und den Speziallfall der Arrays aus Kapitel 14.

Anmerkung 2: Da man Sequenzen als Spezialfall von Abbildungen über dem Intervall $(1..n)$ *auffassen kann, gelten die hier beschriebenen Variationen auch für Sequenzen. Die erste Variante haben wir oben explizit angegeben; sie entspricht der CPO-Sicht der immer länger werdenden Sequenzen. Die zweite Variante entspricht Sequenzen fester Länge mit wachsenden Elementen.*

Programm 10.11 Die Spezifikation *Mappings* als CPO

```
SPECIFICATION Mappings = {                                   -- (siehe Programm 15.3)
                ...
  FUN _ ⊆ _: Map α β × Map α β → Bool   VAR α: Type, β: Type
  DEF m1 ⊆ m2 = ...                                          -- (siehe Programm 15.3)
  FUN _ ∪ _: Map α β × Map α β → Map α β   VAR α: Type, β: Type
  DEF m1 ∪ m2 = ...                                          -- (siehe Programm 15.3)
  FUN _ ⊑ _: Map α β × Map α β → Bool   VAR α: Type, β: Cpo
  DEF m1 ⊑ m2 ⟺ dom(m1) ⊆ dom(m2) ∧ ∀x ∈ dom(m1) • m1.x ≼ m2.x
  FUN _ ⊔ _: Map α β × Map α β → Map α β   VAR α: Type, β: Cpo
  DEF m1 ⊔ m2 = m1 ⊔⊔ m2
}
PROP Mappings ⊆ CompletePartialOrder
                RENAMING (α, ⊥, ≼, ⊔) AS (Map α β, □, ⊆, ∪)
PROP Mappings ⊆ CompletePartialOrder
                RENAMING (α, ⊥, ≼, ⊔) AS (Map α β, □, ⊑, ⊔)
```

Die erste CPO-Variante basiert auf der Teilmap-Relation $\subseteq$ und der zugehörigen Map-Vereinigung $\cup$, die beide in Kapitel 15 diskutiert werden (s. Programm 15.3). Diese Variante funktioniert für beliebige Elementtypen β.

Die zweite Variante „erbt“ die CPO-Eigenschaft des Elementtyps β. Allerdings beschreiben wir hier gleich die verallgemeinerte Form, bei der diese Vererbung mit der Möglichkeit wachsender Indexmengen (gemäß der ersten Variante) verschmolzen wird. Die Ordnung $m_1 \sqsubseteq m_2$ basiert darauf, dass m_2 mindestens so viele Einträge (also definierte Stellen) haben muss wie m_1, und dass m_2 bei jedem gemeinsamen Index das größere Element hat.[6]

Bei der Operation $\overset{\sqcup}{\vee}$ müssen wir diejenige Version von Zip nehmen, die den Wert der einen Map identisch übernimmt, wenn die andere Map an dieser Stelle undefiniert ist. (Schließlich muss die Map „wachsen“.)

[6] Dies entspricht der üblichen Ordnung auf Funktionen, wenn man die fehlenden Elemente von m_1 (also $m_1(x)$ mit $x \in dom(m_2) - dom(m_1)$) als $\bot$ interpretiert.

In der zweiten CPO-Variante hat man für die Berechnung eines bedingten kleinsten Fixpunktes oft eine konstante Indexmenge und als Startwert w eine Funktion, die an allen Stellen den Wert $\bot$ aus der CPO β hat.

Die beiden CPO-Varianten unterscheiden sich auch darin, welche Operatoren für sie jeweils monoton und inflationär sind. Dies ist in den Tabellen 10.4 und 10.5 aufgelistet. (Dabei stammen die Operationen aus den Programmen 10.11 sowie 15.3 und 15.4 in Kapitel 15.) Man beachte, dass sich die beiden Tabellen nur in subtilen Details unterscheiden.

Operation	monoton in ...	inflationär in ...
$m_1 \subseteq m_2$	m_2	
$m_1 \overleftarrow{+} m_2$	m_1	m_2
$m_1 \cup m_2 \quad (*)$	m_1, m_2	m_1, m_2
$m - s$	m	
$dom\ m$	m	
$ran\ m$	m	
$a \in m$	m	

Tab. 10.4: CPO-Eigenschaften wichtiger Map-Operationen für $\subseteq$

Die Annotation $(*)$ der Operation $\cup$ in Tabelle 10.4 sagt aus, dass diese Eigenschaften nur gelten, wenn die Operation wohl definiert ist (s. Kapitel 15).

Operation	monoton in ...	inflationär in ...
$m_1 \sqsubseteq m_2$	m_2	
$m_1 \overleftarrow{+} m_2$	m_1	m_2
$m_1 \sqcup m_2$	m_1, m_2	m_1, m_2
$m - s$	m	
$dom\ m$	m	
$a \in m$	m	
$a \mapsto b$	b	

Tab. 10.5: CPO-Eigenschaften wichtiger Map-Operationen für $\sqsubseteq$

Tupel und Gruppen

Tupel erben die CPO-Eigenschaft ihrer Komponenten. Wir betrachten als phänotypischen Fall Paare.

Man könnte – angelehnt an die Idee der lexikographischen Ordnung bei Strings – auch asymmetrische Ordnungen nehmen, bei denen eine Komponente dominiert und die andere nur dann einen Einfluss hat, wenn die Werte der dominierenden Komponente gleich sind.

Programm 10.12 Die Struktur *Pair*

```
STRUCTURE Pair = {
  TYPE Pair[α: Cpo, β: Cpo] = (1st: α, 2nd: β)
  FUN _ ⊑ _: Pair[α,β] × Pair[α,β] → Bool
  DEF (a,b) ⊑ (a',b') = a ≼ a' ∧ b ≼ b'
  FUN _ ⊔ _: Pair[α,β] × Pair[α,β] → Pair[α,β]
  DEF (a,b) ⊔ (a',b') = (a ⊔ a', b ⊔ b')
}
PROP Pair: CompletePartialOrder RENAMING (α, ⊥, ≼) AS (Pair[α,β], (⊥,⊥), ⊑)
```

Da Paare letztlich nur Abbildungen der (anonymen) Indexmenge $\{1, 2\}$ in $(\alpha \mid \beta)$ sind, entspricht dies genau der Ordnung bei Maps für den Fall konstanter Indexmengen. Aus dem gleichen Grund lassen sich auch Gruppen nach demselben Muster zu CPOs machen, indem sie die CPO-Eigenschaften ihrer Komponententypen erben.

10.5 Beispiel: Lösung von Gleichungssystemen

In vielen Applikationen müssen wir gegebene Gleichungssysteme lösen. Solche Gleichungssysteme haben üblicherweise folgende Form[7]

$$\begin{array}{lcl} x_1 & = & f_1(x_1, ..., x_n) \\ x_2 & = & f_2(x_1, ..., x_n) \\ & \ldots & \\ x_n & = & f_n(x_1, ..., x_n) \end{array} \tag{10.2}$$

Wenn solchen Gleichungssystemen CPOs zugrunde liegen, können wir sie über Fixpunktbildung lösen. Mit der technischen Umsetzung dieser Idee wollen wir uns im Folgenden auseinandersetzen.

Beispiel 10.2 (Fixpunktberechnung)

Zum Einstieg betrachten wir ein einfaches Beispiel. (Später werden wir dieses Beispiel noch einmal aufgreifen und zeigen, aus welcher Anwendung es entsteht.) Es sei ein System von drei Gleichungen über Mengen gegeben:

$$\begin{array}{lcl} x_1 & = & x_1 \cup x_2 \\ x_2 & = & x_2 \cup x_3 \\ x_3 & = & \{i\} \cup \{o\} \end{array} \tag{10.3}$$

[7] Der Einfachheit halber beschränken wir uns auf homogene Gleichungen, in denen alle Variablen den gleichen Typ haben. Die Verallgemeinerung auf heterogene Systeme ist nur technisch hässlicher.

Um die Fixpunktbildung durchzuführen, beginnen wir bei jeder Variablen mit dem Bottom-Element, also der leeren Menge, und berechnen die Kleene-Kette bis sie sich stabilisiert:

$$\begin{array}{l|l|l|l} x_1^{(0)} = \emptyset & x_1^{(1)} = \emptyset & x_1^{(2)} = \emptyset & x_1^{(3)} = \{i,o\} \\ x_2^{(0)} = \emptyset & x_2^{(1)} = \emptyset & x_2^{(2)} = \{i,o\} & x_2^{(3)} = \{i,o\} \\ x_3^{(0)} = \emptyset & x_3^{(1)} = \{i,o\} & x_3^{(2)} = \{i,o\} & x_3^{(3)} = \{i,o\} \end{array} \quad (10.4)$$

Das illustriert, wie die Fixpunktiteration zur Lösung von Gleichungssystemen benutzt werden kann – sofern die Gleichungssysteme über einer CPO formuliert sind.

10.5.1 Repräsentation von Gleichungssystemen

Wie können wir Gleichungssysteme in eine Form bringen, die der Programmierung zugänglich ist? In traditionellen Programmiersprachen ist das mit ziemlichen technischen „Klimmzügen“ verbunden. Aber unsere Entscheidung, Gruppen als *First-class-citizens* zu behandeln, macht diese Aufgabe relativ einfach.

Wir demonstrieren die Technik anhand des obigen Beispiels (10.3). Dieses Gleichungssystem lässt sich einfach in die syntaktische Form einer Gruppe bringen. Das ist in Programm 10.13 gezeigt.

Programm 10.13 Gleichungssystem als Gruppe (mit Rekursion)

```
GROUP tripleSet = {
  DEF x1: Set = x1 ∪ x2
  DEF x2: Set = x2 ∪ x3
  DEF x3: Set = {i} ∪ {o}
}
```

Der Typ dieser Gruppe *tripleSet* ist

TYPE $TripleSet = \{x_1 = Set,\ x_2 = Set,\ x_3 = Set\}$

Damit bleibt aber immer noch die entscheidende Frage offen: *Wie sind diese rekursiven Definitionen zu erklären?* In traditionellen Programmiersprachen wie z. B. OPAL, ML oder HASKELL wären alle drei Mengen undefiniert, weil ihre Berechnung nicht terminieren würde. Deshalb müssen wir hier eine neue Interpretation finden.

Als Erstes erinnern wir uns daran, dass eine Gruppe wie *tripleSet* ein Element ist, das (coalgebraisch) über seine drei Zugriffsfunktionen x_1, x_2 und x_3 (vgl. Kapitel 4) definiert ist. Auf dieser Basis definieren wir eine Funktion f, die Gruppen dieser Art gemäß Gleichung (10.3) in neue Gruppen abbildet:

$$\begin{aligned}&\text{FUN } f\colon \mathit{TripleSet} \to \mathit{TripleSet}\\ &\text{DEF } f(g) = g' \text{ WHERE } g' = \text{GROUP } \{\ x_1 = g.x_1 \cup g.x_2\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad x_2 = g.x_2 \cup g.x_3\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad x_3 = \{i\} \cup \{o\}\ \}\end{aligned}$$

Dabei benutzen wir wieder unser übliche Konvention, diese Zugriffsfunktionen mit Hilfe des „.“-Operators zu schreiben, also $g.x_i$ anstelle von $x_i(g)$.

Mit Hilfe dieser Funktion f können wir unsere Ausgangsgruppe *tripleSet* als rekursive Gleichung formulieren:

$$\text{GROUP } \mathit{tripleSet} = f(\mathit{tripleSet})$$

Diese Rekursion berechnen wir jetzt in der üblichen Weise als Fixpunkt mittels einer entsprechenden Kleene-Kette. Ausgehend von der minimalen Gruppe

$$\text{GROUP } g^{(0)} = \{\, x_1 = \emptyset,\ x_2 = \emptyset,\ x_3 = \emptyset\}$$

bilden wir die Kette

$$g^{(1)} = f(g^{(0)}),\ \ \ldots,\ \ g^{(3)} = f(g^{(2)})$$

Diese Kette entspricht genau der Kette (10.4), die ab $g^{(3)}$ stabil bleibt. Damit ist die Lösung insgesamt

$$\begin{aligned}\text{GROUP } \mathit{tripleSet} = f^3(g^{(0)}) = \{\ & x_1 = \{i, o\}\\ & x_2 = \{i, o\}\\ & x_3 = \{i, o\}\ \}\end{aligned}$$

Diese Überlegungen führen uns auf die programmiertechnische Lösung des Gleichungssystems (10.3), die in Programm 10.14 formuliert ist. Der entscheidende Punkt ist die Umsetzung der Gleichungen (10.3) in die Syntax der Funktion $f_{\mathit{tripleSet}}$.

Programm 10.14 Gleichungssystem als Fixpunktprogramm

```
IMPORT Fixpoint(TripleSet)

FUN f_tripleSet: TripleSet → TripleSet
DEF f_tripleSet(g) = g' WHERE g' = GROUP { x1 = g.x1 ∪ g.x2
                                         x2 = g.x2 ∪ g.x3
                                         x3 = {i} ∪ {o} }
FUN tripleSet: TripleSet
DEF tripleSet = fixpoint(f_tripleSet)
```

Natürlich wäre es wünschenswert, anstelle des Programms 10.14 einfach das ursprüngliche Programm 10.13 nehmen zu können. Das ist möglich, braucht aber einige grundsätzliche Überlegungen, auf die wir im folgenden Abschnitt 10.5.4 gleich eingehen werden.

10.5.2 Optimierung

Wie wir weiter oben gezeigt haben, lassen sich Gleichungssysteme so umbauen, dass sie als Join mehrerer Basisfunktionen dargestellt werden. Wenn eine Funktion f derart als *Join* von Funktionen gegeben ist, lässt sich die Fixpunktbildung genauso berechnen. Sei also

$$f(x) = f_1(x) \sqcup \ldots \sqcup f_n(x) \tag{10.5}$$

Dann können wir folgendes Programm benutzen (das wir hier sehr informell notieren):

$$\begin{array}{l}
\textsc{def}\ \mathit{iterate}(f_1, \ldots, f_k)(s) = \\
\quad \textsc{let}\ \mathit{workset}_1 = \Delta_1(s, f_1(s)) \\
\qquad \ldots \\
\qquad \mathit{workset}_k = \Delta_k(s, f_k(s)) \\
\quad \textsc{in} \\
\qquad \textsc{if}\ \forall i \bullet \mathit{workset}_i = \emptyset\ \textsc{then}\ s \\
\qquad \textsc{if}\ \exists i \bullet \mathit{workset}_i \neq \emptyset\ \textsc{then}\ \textsc{let}\ d = \mathit{arb}(\mathit{workset}_i) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsc{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{iterate}(f)(s \oplus_i d) \\
\qquad\qquad\qquad\qquad\quad \textsc{fi}
\end{array}$$

10.5.3 Nochmals: Grammatiken und Parser

Es gibt viele Anwendungsbereiche, in denen man Gleichungssysteme lösen oder transitive Hüllen berechnen muss. Ein Beispiel aus dem Bereich der Graphentheorie haben wir am Anfang dieses Kapitels präsentiert. Ein anderes Gebiet, aus dem interessante Aufgabenstellungen kommen, ist der Bereich des Compilerbaus. Hier gibt es insbesondere im Bereich der so genannten Kontroll- und Datenfluss-Analyse zahlreiche Situationen, in denen man Gleichungssysteme über CPOs lösen muss, meistens über dem Typ *Set* α.

Wir betrachten hier ein Beispiel aus dem Parsing-Teil des Compilers, nämlich einen Teilaspekt der Frage, ob eine Grammatik die so genannte LL(1)-Eigenschaft besitzt. Da wir hier aber nicht Übersetzerbau betreiben wollen, sondern nur Illustrationen für unsere Fixpunkt-Techniken suchen, nehmen wir uns die Freiheit, das Problem etwas zu vereinfachen.

Beispiel 10.3 (First-Mengen)

Wir betrachten die Berechnung der so genannten *First-Mengen*, allerdings eingeschränkt auf Epsilon-freie Grammatiken, also auf Grammatiken, deren Produktionen keine leeren rechten Seiten haben. Ein klassisches Beispiel für eine solche Grammatik ist die der arithmetischen Ausdrücke über „+“ und „∗“. Zur besseren Lesbarkeit folgen wir hier der Konvention, Terminalzeichen nicht in Apostrophe einzuschließen, sondern durch Unterstreichung zu kennzeichnen; das Terminalzeichen *ide* steht für beliebige Identifier.

$$\mathcal{G} = \boxed{\begin{array}{lcl} E & \to & E \ \underline{+} \ T \\ E & \to & T \\ T & \to & T \ \underline{*} \ F \\ T & \to & F \\ F & \to & \underline{ide} \\ F & \to & \underline{(} \ E \ \underline{)} \end{array}} \tag{10.6}$$

Die First-Mengen ordnen jeder Produktion einer Grammatik die Menge derjenigen Terminalzeichen zu, mit denen die zugehörigen Sätze anfangen können. Bei Epsilon-freien Grammatiken genügt es, die First-Mengen nur für Nichtterminalzeichen zu berechnen; wir suchen also die Mengen

$$first(N) = \{\, t \in T \mid \exists u \in V^* : \ N \stackrel{*}{\Rightarrow} u \wedge t = ft(u) \,\} \tag{10.7}$$

Diese First-Mengen ergeben sich in unserer Beispielgrammatik $\mathcal{G}$ aus (10.6) als minimale Lösungen des folgenden (Un)Gleichungssystems, wobei wir kurz i für das Terminalzeichen $\underline{ide}$ und o für das Terminalzeichen $\underline{(}$ schreiben:

$$\begin{array}{l} ft(E) \supseteq ft(E) \\ ft(E) \supseteq ft(T) \\ ft(T) \supseteq ft(T) \\ ft(T) \supseteq ft(F) \\ ft(F) \supseteq \{i\} \\ ft(F) \supseteq \{o\} \end{array} \tag{10.8}$$

Die Lösung dieses Systems ist gleichwertig zu der Lösung des folgenden, einfacheren Systems.[8]

$$\begin{array}{l} ft(E) = ft(E) \cup ft(T) \\ ft(T) = ft(T) \cup ft(F) \\ ft(F) = ft(F) \cup \{i\} \cup \{o\} \end{array} \tag{10.9}$$

Wie wir weiter vorne gesehen haben, führen diese Gleichungen (bis auf Umbenennung) direkt auf Programm 10.14.

Anmerkung: Wie wir in Kapitel 3 angesprochen haben, nützen uns die First-Mengen bei der Beispielgrammatik $\mathcal{G}$ aus (10.6) zunächst relativ wenig, weil die Grammatik linksrekursiv ist und deshalb a priori nicht LL(1) sein kann. Deshalb muss man die Grammatik entsprechend transformieren. Aber die Berechnung der First-Mengen kann – wenn auch mit leichten Adaptionen wegen der entstehenden Epsilon-Produktionen – auch für die transformierte Grammatik benutzt werden.

Der Weg von der Grammatik 10.6 zum Programm 10.14 braucht noch relativ viel mathematische Arbeit durch den Programmierer. In Kapitel 3.2 hatten

[8] Das sind gerade die Gleichungen, die wir am Anfang des Abschnitts 10.5 zur Illustration herangezogen haben.

wir es dagegen bei Parsern geschafft, durch die Bereitstellung geeigneter Operatoren die Grammatik selbst nahezu unverändert als Parser zu nehmen. Lässt sich etwas Ähnliches auch bei der Berechnung der First-Mengen erreichen?

Die Antwort wird in Programm 10.15 gegeben. Wir benutzen hier – genau wie bei Parsern – Operatoren wie „ ; “ und „ | “, um die Produktionen der Grammatik direkt in programmiersprachliche Ausdrücke umzuschreiben. Allerdings kommen ihre Definitionen jetzt aus einer anderen Struktur, nämlich aus der Struktur *MetaFirst*, die in Programm 10.16 definiert ist.

Programm 10.15 Grammatik als Gleichungssystem für First-Mengen

```
IMPORT Fixpoint(TripleSet)
IMPORT MetaFirst
TYPE FirstSet = { E = Set Terminal, F = Set Terminal, T = Set Terminal}
FUN f_firstset: FirstSet → FirstSet
DEF f_firstset(g) = g' WHERE g' = GROUP { E = (g.E ; plus ; g.T)  |  g.T
                                          T = (g.T ; star ; g.F)  |  g.F
                                          F = ide  |  (open ; g.E ; close) }
FUN firstset: FirstSet
DEF firstset = fixpoint(f_firstset)
```

Der Operator „ | “ stellt bei einer Grammatik die Auswahl zwischen zwei Teilsprachen dar; deshalb muss man die entsprechenden First-Mengen vereinigen. Der Operator „ ; “ steht für die sequenzielle Komposition zweier Teilsprachen A und B; wegen unserer Annahme der Epsilon-Freiheit muss man deshalb nur die First-Menge von A nehmen. Terminalzeichen sind offensichtlich jeweils ihre eigene First-Menge. Der Vollständigkeit halber nehmen wir auch semantische Aktionen hinzu (die wir zur Vereinfachung in Programm 10.15 weggelassen haben); diese generieren keine Zeichen und haben deshalb auch leere First-Mengen.

Wenn die Grammatik Epsilon-Produktionen enthält, werden diese Definitionen etwas komplexer. Üblicherweise ordnet man solchen leeren Produktionen ein Pseudo-Terminalzeichen „ε“ zu, sodass z. B. bei semantischen Aktionen anstelle der leeren Menge dieses Pseudo-Terminal abgeliefert wird:

```
DEF â = { ε}
```

Auch der Sequenz-Operator „ ; “ muss jetzt angepasst werden: Falls A ein „ε“ enthält, muss auch die First-Menge von B betrachtet werden.

Diese Ansätze sehen erfreulich elegant aus, aber wir wollen nicht verschweigen, dass es noch Probleme und Komplikationen gibt.

- Beim Vorliegen von Epsilon-Produktionen braucht man neben den First-Mengen auch die so genannten Follow-Mengen (auf die wir hier nicht näher eingehen). Diese lassen sich nach einem ähnlichen Fixpunktschema

Programm 10.16 Operatoren für First-Mengen

```
STRUCTURE MetaFirst = {
  FUN _ | _: Set Terminal × Set Terminal → Set Terminal
  DEF A | B = A ∪ B
  FUN _ ; _: Set Terminal × Set Terminal → Set Terminal
  DEF A ; B = A
  FUN _: Terminal → Set Terminal
  DEF (t: Terminal): Set Terminal = { t }
  FUN ^_: Action → Set Terminal
  DEF ^a = ∅
}
```

aus der Grammatik ableiten. Allerdings sind dann die First- und Follow-Mengen wechselseitig voneinander abhängig, was eine gemeinsame Fixpunktbildung nötig macht. Außerdem gibt es technische Probleme mit der Benennung der jeweiligen Mengen (die jetzt ja Gruppenselektoren sind).

- Die meisten Programmiersprachen lassen Gruppen nicht als *First-class citizen* zu. Dann kann man unsere obigen Techniken zwar immer noch anwenden, aber man muss jetzt auf Maps übergehen, die die Nichtterminalsymbole als Indexmenge haben. Das führt im Beispiel der Grammatik (10.6) auf folgendes Design:

  l

  TYPE $Symbol = \{ E,\ T,\ F \}$

  FUN $f_{firstset}$: $Map\ Symbol\ (Set\ Terminal) \to Map\ Symbol\ (Set\ Terminal)$
  DEF $f_{firstset}(m) = m'$ WHERE
  $$m' = \text{GROUP}\ \{\ E \mapsto (m(E)\ ;\ plus\ ;\ m(T))\ \mid\ m(T)$$
  $$T \mapsto (m(T)\ ;\ star\ ;\ m(F))\ \mid\ m(F)$$
  $$F \mapsto ide\ \mid\ (open\ ;\ m(E)\ ;\ close)\ \}$$

  Hier benutzen wir die CPO-Variante für Maps, die auf konstanten Indexmengen und der Vererbung der CPO-Eigenschaft des Elementtyps basiert.

10.5.4 Ein Gedankenexperiment: Anpassbare Rekursion

Wenn wir das Beispiel aus dem vorigen Abschnitt nochmals betrachten, dann lässt sich die Essenz eigentlich in folgender rekursiver Definition erfassen:

$$\begin{aligned}\text{GROUP}\ g = \{\ &\text{DEF}\ E\ =\ (E\ ;\ plus\ ;\ T\ ;\ \widehat{e_1})\ \mid (T\ ;\ \widehat{e_2})\\ &\text{DEF}\ T\ =\ (T\ ;\ star\ ;\ F\ ;\ \widehat{t_1})\ \mid (F\ ;\ \widehat{t_2})\\ &\text{DEF}\ F\ =\ (ide\ ;\ \widehat{f_1})\ \mid (open\ ;\ E\ ;\ close\ ;\ \widehat{f_2})\ \}\end{aligned}$$

Diese Gruppe liefert bei einer geeigneten Interpretation den Parser (wenn wir von dem Problem seiner Nichtterminierung wegen der Linksrekursion einmal großzügig absehen). Dies wurde in Kapitel 3 demonstriert.

In einer anderen Interpretation der Operatoren führt die gleiche Gruppe zur Berechnung der First-Mengen. Allerdings hatten wir dazu – wie in den vorausgegangenen Abschnitten gezeigt – etwas technischen Überbau hinzufügen müssen.

Damit liegt die Spekulation nahe, ob sich diese Unterschiede nicht systematisch so erfassen lassen, dass man die Gruppe tatsächlich identisch notieren kann. Dazu müsste man in die Sprache einen expliziten Fixpunktoperator aufnehmen, der mit der Art der CPO parametrisiert werden kann:

```
GROUP firstset =  FIX[Set] {...}          -- für die First-Mengen
GROUP parser   =  FIX[LANGUAGE] {...}     -- für den Parser
```

Für Daten-CPOs wie *Set* würde die Fixpunkt-Berechnung genommen werden, die in diesem Kapitel entwickelt wurde. Für die „Sprach-CPO“ würde der klassische Rekursionsmechanismus gewählt werden, wie er in Compilern standardmäßig eingebaut ist. (Dabei sollte man die letztere Version als Default nehmen, weil sie wesentlich häufiger auftritt; hier sollte man auch auf die explizite Angabe des Operators FIX verzichten dürfen.)

Man könnte diese Idee noch etwas weiter treiben und bei der Sprach-CPO Variationen für Call-by-value (eager) und Call-by-need (lazy) zulassen.

Aber das ist alles Spekulation ...

11

Beispiel: Monaden

> *... dass man ... beginnend mit einer exakten ... Darlegung der Monadenlehre, prüfe, ob entweder die Monaden durch unanfechtbare Argumente gründlich widerlegt und zerstört werden können, oder ... die Monaden zu beweisen ... und daraus eine verständliche Erklärung der hauptsächlichsten Phänomene des Universums ... abzuleiten.*
>
> *Preisaufgabe der Berliner Akademie für das Jahr 1747*

Zum Abschluss des Themenkomplexes „Typisierung“ wollen wir uns noch an ein besonders kniffliges Beispiel wagen. Der Hintergrund dafür ist ein Konzept, das seit einiger Zeit durch die Welt der Funktionalen Programmierung geistert: *Monaden*. Obwohl es etwas sperrig ist, hat dieses Konzept in den letzten Jahren eine immer größere Bedeutung in diversen Applikationen erlangt.

Eigentlich sind Monaden ein Begriff aus der Kategorientheorie [91, 45, 48], einem besonders esoterischen Teil der Mathematik.[1] In die Welt der Informatik geriet das Konzept erstmals im Rahmen von theoretischen Untersuchungen zur Semantik von Programmiersprachen – und dort erfüllte es auch einen guten und seinem Wesen angemessenen Zweck.

Doch populär wurde das Wort in einem ganz anderen Kontext. Es gibt eine Programmiertechnik, die in funktionalen Sprachen seit langem verwendet wird, um Ein-/Ausgabe zu programmieren. (Im OPAL-Compiler wird diese Technik z. B. als *Commands* seit Anfang der 80er Jahre standardmäßig zur Ein-/Ausgabe verwendet.) Irgendwann hat jemand die Beobachtung gemacht, dass diese Technik zufällig die Monadenaxiome erfüllt – und dass „Monade“ ein schickes Wort ist, das man vorzüglich als PR-Maßnahme (miss)brauchen kann. Und jetzt wird eben alles, was in diesem Teil der Programmierung

[1] Noch früher wurde das Wort von Leibniz benutzt, der unter dem Begriff Monaden (gr.: *monas*: Einheit) in die Philosophie das einführen wollte, was in der Physik ursprünglich unter dem Namen Atom verstanden wurde: die kleinste, nicht weiter teilbare Einheit. Diese Sicht geht letztlich bereits auf die Pythagoreer zurück.

passiert, mit dem Schlagwort Monaden bezeichet, auch wenn die zugehörige Theorie nirgends wirklich benutzt wird.

Inzwischen haben sich noch einige weitere interessante Programmiertechniken rund um die Monadenidee etabliert, z. B. Variationen des Map-Filter-Reduce-Ansatzes, monadische Parser, so genannte *Arrow types* [83] usw.

Wir werden die (ungemein nützliche) Programmiertechnik, die unter dem Schlagwort Monaden läuft, erst in den Kapiteln 17 bis 20 genauer einführen und auf die Programmierung von Ein-/Ausgabe, GUIs, parallelen Agenten etc. anwenden. Jetzt wollen wir „nur“ ein Vorspiel zu ihrer theoretischen Fundierung machen. Denn mit Hilfe der Typisierungshierarchie aus Kapitel 9 brauchen wir Monaden nicht mehr ad hoc einzuführen („*yet another feature*“), sondern können sie ganz systematisch aufbauen – und zwar orientiert an ihren mathematischen Grundlagen.

Um eines aber von Anfang an klar zu machen: Wir versuchen *nicht*, die mathematische Kategorientheorie im Rahmen unserer Programmiersprache nachzubauen. Vielmehr wollen wir zeigen, dass gewisse Teilaspekte unserer Sprachkonstrukte „zufällig“ Monaden sind.

11.1 Kategorien, Funktoren und Monaden

Monaden sind ein ziemlich abstraktes Konzept. Deshalb werden wir uns bei der formalen Beschreibung sehr kurz fassen und uns primär an illustrativen und wichtigen Beispielen orientieren. Die zugrunde liegende Mathematik werden wir nur in Form von Anmerkungen skizzieren.

Für die systematische Einführung des Begriffs Monade brauchen wir zwei weitere Konzepte aus der Kategorientheorie: Monaden sind spezielle *Funktoren*. Und beides baut auf dem Begriff der *Kategorie* auf.

11.1.1 Typklassen als Kategorien

Im Wesentlichen ist jede Typklasse eine ***Kategorie***. Denn die mathematische Definition verlangt dafür nur die Existenz der Identitätsfunktion und der Funktionskomposition. Letztere muss allerdings bzgl. der Typklasse abgeschlossen sein.

Für unsere weitere Arbeit ist diese Sichtweise allerdings unnötig filigran und damit zu aufwendig. Deshalb betrachten wir im Folgenden nur die größte Typklasse, nämlich die Klasse `Type` aller Typen. Dies wird in der Spezifikation von Programm 11.1 festgelegt.

Anmerkung: In der Mathematik sind Kategorien eigentlich alles, was aus Objekten *und* Pfeilen *zwischen diesen Objekten besteht. Einzige Bedingungen sind die Existenz eines Identitätspfeils auf jedem Objekt und die Existenz einer assoziativen Komposition von Pfeilen. Beides wird in funktionalen Sprachen von Typen (Objekten) und Funktionen (Pfeilen) trivialerweise erfüllt.*

Programm 11.1 Typklassen als Kategorien

```
SPECIFICATION TypesAsCategory = {
  FUN id: [α: Type] ↦ α → α
  FUN _ ∘ _: [α: Type, β: Type, γ: Type] ↦ (β → γ) × (α → β) → (α → γ)
  PROP id(x) = x                                   -- Identität
  PROP f ∘ id = id ∘ f = f                         -- Identität ist neutral
  PROP h ∘ (g ∘ f) = (h ∘ g) ∘ f                   -- Assoziativität
}
```

11.1.2 Polymorphe Typen als Funktoren

In der Mathematik sind ***Funktoren*** generell Abbildungen von Kategorien in Kategorien. In unserer programmiersprachlichen Verwendung sind es Abbildungen von Typen auf Typen, also das, was wir als generische oder polymorphe Typen bezeichnet haben (wie z. B. *Seq* α). Dazu kommt ein Operator, der Funktionen der einen Kategorie in Funktionen der anderen Kategorie abbildet; wir schreiben ihn (nicht zufällig) als „ $*$ “. Das wird in der Spezifikation von Programm 11.2 ausgedrückt.

Programm 11.2 Die Spezifikation von Funktoren

```
SPECIFICATION Functor = {
  TYPE F: Type → Type                              -- generischer Typ
  FUN _*: [α: Type, β: Type] ↦ (α → β) → (F α → F β)   -- Map-Operator
  PROP id* = id                                    -- Overloading beachten!
  PROP (g ∘ f)* = (g*) ∘ (f*)
}
```

In dieser Spezifikation werden die Eigenschaften von Funktoren festgelegt: Der Operator „ $*$ “ überführt die Identität der einen Kategorie in die Identität der anderen Kategorie und er distribuiert über die Funktionskomposition. (Es ist kein Zufall, dass das genau die Eigenschaften z. B. des Map-Operators sind.)

Anmerkung 1: In HASKELL *werden Funktoren im Standard Prelude eingeführt in der (leicht an unseren Stil adaptierten) Form*

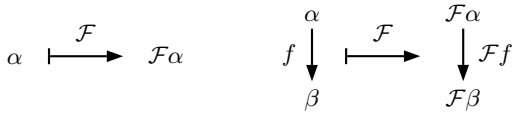
```
class Functor F where
  fmap :: (α → β) → (F α → F β)
```

Die Applikation wird dann als entsprechende Instanz definiert, z. B.

```
instance Functor Seq where
  fmap = *
```

Anmerkung 2: In der Mathematik wird ein Funktor üblicherweise in Form von zwei Abbildungen dargestellt, einer Objektabbildung *und einer* Pfeilabbildung. *Der*

Zusammenhang wird in einem Diagramm der folgenden Bauart dargestellt (wobei der Pfeiloperator nicht als $f *$ sondern als $\mathcal{F}f$ geschrieben wird):



Objektabbildung Pfeilabbildung

11.1.3 Monaden

Kategorien und Funktoren sind – im Kontext funktionaler Sprachen – eigentlich sehr einfache Konzepte (wenn man von den etwas merkwürdigen Namen absieht). Das eine sind de facto Typklassen, das andere generische Datentypen zusammen mit dem Map-Operator. Demgegenüber sind die Monaden etwas sperriger.

In der Mathematik sind Monaden spezielle Funktoren, für die gewisse *natürliche Transformationen* (s. unten) existieren. Diese Transformationen manifestieren sich im Wesentlichen in drei Funktionen (vgl. Programm 11.3): Die erste – *lift* – ist einfach, sie stellt eine Einbettung des Basistyps in den generischen Typ dar (z. B. die Bildung der einelementigen Sequenz aus einem Element). Die beiden anderen – *flatten* und & – sind dagegen komplexer und werden erst im Zusammenhang mit den folgenden Beispielen verständlich werden.

Definition (Monade)

Generell lassen sich Monaden einfach charakterisieren: Eine ***Monade*** ist ein polymorpher Typ $M\ \alpha$ zusammen mit vier speziellen Funktionen $*$, *lift*, *flatten* und & (s. Programm 11.3).

Wir haben in Programm 11.3 auch die relevanten Eigenschaften von Monaden mit aufgelistet, obwohl die meisten davon im Augenblick eher mystisch wirken.

Aber etwas kann man bereits sehen: Die letzten drei Eigenschaften zeigen, dass die Operatoren nicht unabhängig voneinander sind. So lässt sich z. B. & durch *flatten* und $*$ ersetzen, *flatten* alleine durch &, und schließlich $*$ durch & und *lift*. Deshalb haben wir die Eigenschaften auch in Form von Default-Definitionen (vgl. Kapitel 9) angegeben.

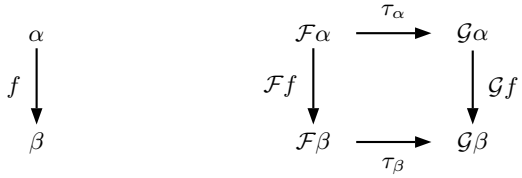
Diese Abhängigkeiten werden uns in den Beispielen helfen. In manchen Fällen ist nämlich *flatten* intuitiv sofort klar, aber & rätselhaft, und in anderen Fällen ist es genau umgekehrt.

Anmerkung: Wenn man Bücher der Kategorientheorie liest, dann ist der nächste Schritt nach der Einführung von Funktoren die Abbildung von Funktoren auf Funktoren. Eine solche Abbildung $\tau : \mathcal{F} \xrightarrow{\bullet} \mathcal{G}$ heißt ***natürliche Transformation****, wenn*

Programm 11.3 Die Spezifikation von Monaden

```
SPECIFICATION Monad = {
  EXTEND Functor RENAMING F AS M                    -- TYPE M = (Type → Type)
  FUN lift: [α: Type] ↦ α → M α
  FUN flatten: [α: Type] ↦ M (M α) → M α
  FUN _&: [α: Type, β: Type] ↦ M α → (α → M β) → M β
  FUN _&: [α: Type, β: Type] ↦ M α → M β → M β
  PROP (f∗) ∘ lift = lift ∘ f
  PROP flatten ∘ lift = id
  PROP flatten ∘ (lift∗) = id
  PROP (f∗) ∘ flatten = flatten ∘ ((f∗)∗)
  PROP lift a & f = f a
  PROP m & lift = m
  PROP (m & f) & g = m & (λ a • f a & g)
  DEF (m1 & m2) = m1 & (λ a • m2)                   -- Default-Definition
  DEF m & f = flatten(f ∗ m)                        -- Default-Definition
  DEF flatten(mm) = mm & id                         -- Default-Definition
  DEF f ∗ m = m & (lift ∘ f)                        -- Default-Definition
}
```

sie mit der Funktor-Operation „ ∗ “ kompatibel ist. In entsprechenden Mathematikbüchern wird das gerne durch ein Diagramm folgender Bauart illustriert:



Das ist für Nicht-Mathematiker nicht allzu hilfreich, bekommt aber einen etwas intuitiveren Zugang, wenn wir über Beispiele nachdenken:

- τ: *Seq* $\alpha \xrightarrow{\bullet}$ *Set* α
- τ: *Array* $\alpha \xrightarrow{\bullet}$ *Seq* α
- τ: *Matrix* $\alpha \xrightarrow{\bullet}$ *Array* α

In allen diesen Beispielen – deren intuitives Verständnis wir hier voraussetzen – gibt es eine natürliche Strukturänderung, wobei der ∗ -Operator mit diesen Änderungen kompatibel ist.

In der Kategorientheorie werden ***Prämonaden*** *als spezielle Funktoren eingeführt. Damit ein Funktor $\mathcal{P}$ eine Prämonade ist, muss es eine natürliche Transformation vom Identitätsfunktor auf $\mathcal{P}$ geben. (Das ist einer der Fälle, in denen es aus technischen Gründen nützlich ist, den Identitätsfunktor zu haben.) Wir haben diese Operation lift genannt.*

Eine Prämonade $\mathcal{M}$ ist eine ***Monade****, wenn es eine natürliche Transformation von $\mathcal{M} \circ \mathcal{M}$ nach $\mathcal{M}$ gibt. Unglücklicherweise gibt es zwei gleichwertige Formen, in*

denen man dies realisieren kann. Die eine Variante ist „natürlicher“ im Falle von Funktoren wie Seq und Maybe; wir haben sie flatten genannt. Die andere Variante passt intuitiver zu Funktoren wie Machine; wir haben sie als & geschrieben.

11.2 Beispiele für Monaden

Die Spezifikationen der Funktoren und Monaden samt ihrer zugehörigen Funktionen wirken reichlich abgehoben und mystisch. Sie lassen sich nur verstehen, wenn man sie mit Hilfe einiger charakteristischer Beispiele illustriert. Dies soll im Folgenden geschehen. Dabei wird sich zeigen, dass einige unserer bekannten Datenstrukturen überraschenderweise auch Monaden sind. Wir werden aber auch neue, sehr spezifische Strukturen sehen, die sich später als fundamental für einige Anwendungsgebiete erweisen werden.

11.2.1 Sequenzen als Monaden

Generische Datenstrukturen wie Sequenzen, Mengen, Bäume etc. zusammen mit der fundamentalen Operation *Map* (*Apply-to-all*) sind Funktoren. Wenn dann noch die Bildung einelementiger Sequenzen und das *Flattening* von Sequenzen von Sequenzen hinzukommen, dann erhalten wir Monaden. Dieser Aspekt wird in Programm 11.4 demonstriert. Dabei wiederholen wir aus Gründen der Lesbarkeit die Signaturen, obwohl sie sich aus der Typisierung bereits ergeben.

Programm 11.4 Generische Sequenz als Monade

```
STRUCTURE Sequence: Monad RENAMING M AS Seq = {
  TYPE Seq[α] = _ ·: _(ft = α, rt = Seq α) | Empty        -- monadischer Typ
  FUN _*: (α → β) → (Seq α → Seq β)                       -- Apply-to-all
  DEF f * (a ·: rest) = (f a) ·: (f * rest)
  DEF f * ◊ = ◊
  FUN lift: α → Seq α                                     -- singleton
  DEF lift a = a ·: ◊
  FUN flatten: Seq (Seq α) → Seq α                        -- conc-reduce
  DEF flatten = ++ /
  FUN _&: Seq α → (α → Seq β) → Seq β                     -- apply-and-conc
  DEF s & f = ++ / f * s
}
```

Interessant ist hier, dass der *Apply-to-all*-Operator „ $*$ “ eine so fundamentale Rolle spielt; denn er macht den generischen Typ *Seq* α zum Funktor. Und Funktoren sind das fundamentalste Konzept der Kategorientheorie.

Mit *lift* haben wir eine weitere Standardfunktion, nämlich die Einbettung des Basistyps α in den Typ *Seq* α: Aus einem Element x wird die einelementige Sequenz $\langle x \rangle$. Und *flatten* bildet so etwas wie das Gegenstück dieser Einbettung; es reduziert iterierte Anwendungen der Sequenzbildung auf eine einfache Sequenzbildung: Eine Sequenz von Sequenzen wird durch Konkatenation zu einer flachen Sequenz gemacht.

Mit diesen beiden zusätzlichen Operationen *lift* und *flatten* haben wir aus dem Funktor eine Monade gemacht. Die Operation „ & “ fällt etwas aus dem Rahmen, weil sie intuitiv nicht naheliegend ist. Deshalb haben wir nur die Default-Definition aus der Monadenspezifikation in Programm 11.3 instanziiert. Anwendungen sind nicht allzu häufig, aber es gibt sie. Wenn eine Operation f aus einem Element des Typs α eine ganze Sequenz von β-Elementen erzeugt, dann führt $f * s$ auf eine Sequenz von Sequenzen. Diese muss anschließend per *flatten* wieder zu einer flachen Sequenz gemacht werden.

Bei anderen generischen Datentypen ist die Situation ähnlich wie bei den Sequenzen. Bei Mengen sieht alles analog aus, bei Arrays muss man dagegen die Umindizierung bei *flatten* berücksichtigen. Trickreicher ist das bei Bäumen, denn hier ist der Effekt von *flatten* – die Umwandlung eines Baums von Bäumen in einen einzigen Baum – nicht ganz so offensichtlich. (Wir überlassen die Definition dem interessierten Leser als Übung.)

11.2.2 *Maybe* als Monade

Der generische Typ *Maybe* α kann ebenfalls als Monade dargestellt werden. Seine Definition ist in Programm 11.5 angegeben. Die Operation $f *$ ist im

Programm 11.5 Der generische Typ *Maybe* als Monade

```
STRUCTURE Maybe: Monad RENAMING M AS Maybe = {
  TYPE Maybe α = α | {fail}                          -- generischer Typ

  FUN _*: (α → β) → (Maybe α → Maybe β)              -- Map
  DEF f * a = f a                                    -- Castings implizit!
  DEF f * fail = fail

  FUN lift: α → Maybe α                              -- Up-Casting
  DEF lift a = a AS Maybe α

  FUN flatten: Maybe Maybe α → Maybe α               -- einmal Down-Casting
  DEF flatten (a: α) = (a: Maybe α)                  -- Castings implizit!
  DEF flatten (fail: Maybe α) = (fail: Maybe α)
  DEF flatten (fail: Maybe Maybe α) = (fail: Maybe α)

  FUN _&: Maybe α → (α → Maybe β) → Maybe β
  DEF a & f = f a                                    -- Castings implizit!
  DEF fail & f = fail
}
```

Wesentlichen f zusammen mit den entsprechenden Up- und Down-Castings. Wenn man die Castings in der ersten Gleichung explizit macht, kann man die Funktoreigenschaft direkt sehen:

$$\text{DEF } f * (a \text{ AS } \mathit{Maybe}\ \alpha) = ((f\ a) \text{ AS } \mathit{Maybe}\ \beta)$$

Dass wir $f *$ auf *fail* als Identität erklären, hat nichts mit den Funktoreigenschaften zu tun, sondern ist eine (vernünftige) Design-Entscheidung.

Bei *flatten* müssen wir drei Fälle unterscheiden. Wenn ein Element $a\colon \alpha$ zweimal in ein *Maybe* gecastet wurde, wird dies zu einer einfachen Einbettung reduziert. (Man beachte, dass auf der linken Seite zwei automatische Upcasts erfolgen.) Wenn ein *fail* vom Typ *Maybe* α ein weiteres Mal in ein *Maybe* eingebettet wurde, wird daraus durch *flatten* wieder das alte *fail*. (Auch hier erfolgt links ein automatisches Upcast.) Und das *fail* von *Maybe Maybe* α selbst wird durch *flatten* auf das *fail* von *Maybe* α heruntergedrückt.

Der Operator $a \,\&\, f$ ist sehr interessant: Er liefert im Wesentlichen eine Postfix-Notation der Funktionsapplikation (analog zum Punkt-Operator $x.f$, den wir in früheren Kapiteln zur Simulation objektorientierter Notationen benutzt haben). Aber der &-Operator schließt noch eine Art von implizitem Exception-Handling mit ein. Wenn das Argument *fail* war, dann wird der Fehler propagiert; ansonsten wird f angewandt (was auf eine neue Exception führen kann).

11.2.3 Automaten als Monaden („Zustands-Monaden“)

Das Originalbeispiel, mit dem die Monaden-Euphorie in der funktionalen Programmierung begann, ist die so genannte ***Zustands-Monade*** (engl.: *State-Monade*). Wie so vieles in der Monadenbegeisterung der funktionalen Programmierer, ist aber auch dieser Begriff ein kleines bisschen missraten. Denn eigentlich sollte man hier richtiger von *Automaten-Monade* oder *Maschinen-Monade* sprechen. (Weshalb, werden wir gleich sehen.) Weil der Begriff Zustands-Monade sich aber fest in der Literatur etabliert hat, werden wir ihn beibehalten.

Ein wichtiges Charakteristikum funktionaler Programmiersprachen ist, dass sie von Zeit und Zustand unabhängig sind. Aber spätestens bei der Interaktion mit der Umwelt – insbesondere dem Benutzer – kommt der Aspekt „Zeit“ der realen Welt zum Tragen.[2] Um diesen Effekt so weit wie möglich aus der heilen Welt der Funktionen herauszuhalten, wird der Zustand in einen (verborgenen) Typ *State* gekapselt. Die Elemente von *State* nennen wir „Zustände“. Um diesen Typ *State* herum wird eine parametrisierte Struktur gebaut, die „zufällig“ eine Instanz der Spezifikation *Monad* ist. Wichtiger ist aber: Diese Struktur beschreibt letztlich nichts anderes als einen guten alten Bekannten der Informatik: *Automaten*. Da wir aber nicht die ganze Automatentheorie meinen, sprechen wir lieber von *Maschinen*.

[2] Genauer wird das Phänomen „Zeit“ bei der Funktionalen Programmierung in Kapitel 17 analysiert werden; hier erfolgt nur eine theoretische Vorbereitung.

Programm 11.6 Der generische Typ *Machine* als Monade

```
STRUCTURE Machine: Type → Monad RENAMING (M, lift) AS (Com, yield)
DEF Machine(State) = {
  TYPE Com α = (observe = State → α, evolve = State → State)
  FUN _∗: (α → β) → (Com α → Com β)                    -- Map
  DEF f ∗ (obs, ev) = (f ∘ obs, id ∘ ev)
  FUN yield: α → Com α                                  -- lift
  DEF yield a = (K a, id)
  FUN flatten: Com Com α → Com α                        -- iterierte Ausführung
  DEF flatten cc = cc & id
  FUN _&: Com α → (α → Com β) → Com β                   -- sequ. Komposition
  DEF (obs, ev) & f = (f ∘ obs) S ev
                      WHERE
                      (f S g)x = (f x)(g x)             -- S-Kombinator
  FUN _&: Com α → Com β → Com β                         -- Variante (Kurzform)
  DEF m1 & m2 = m1 & (K m2)                             -- konstante Funktion
  }
```

Die Arbeit von Maschinen besteht in der Ausführung von Instruktionen. Diese Instruktionen haben üblicherweise zwei Effekte: Zum einen bewirken sie einen internen Zustandsübergang, zum anderen liefern sie nach außen sichtbare Resultate. Diese Situation wird in dem Typ *Com* (für *Command*) repräsentiert.

Definition (Kommando)

Ein (monadisches) ***Kommando*** ist ein Paar von Funktionen. Die erste ist eine externe Beobachtungsfunktion *observe*, die aus dem inneren Zustand einen extern sichtbaren Wert extrahiert. Die zweite ist eine interne Transformation *evolve*, die den internen Zustand in einen neuen Zustand überführt.

Programm 11.6 enthält die Definition der Struktur *Machine*. Der Typ *Com* ist generisch; sein Parameter ist der Typ der extern beobachtbaren Werte. Insgesamt ist der *Typ Com* also zweifach parametrisiert: einmal mit dem Strukturparameter *State* und ein zweites Mal mit seinem eigenen Parameter α. Am besten sieht man das an einem Beispiel. In dem später noch zu betrachtenden Spezialfall der Ein-/Ausgabe-Monade hat ein Kommando wie *readInt*(*file*) zwei Aspekte. Zum einen ändert es den internen Zustand der Datei, indem es den Lesezeiger weiter setzt, zum anderen liefert es aber einen extern sichtbaren Wert von Typ *Int*.

Die Operation „ $*$ “ wendet wie üblich die Funktion f auf ein Kommando *cmd* an. Das heißt hier: f wird auf den sichtbaren Wert angewandt. Sei also s ein Zustand, dann gilt $(f * cmd)(s) = (f(cmd.observe(s)), cmd.evolve(s))$.

Die Operation *lift* wird bei dieser Monade meistens als *yield* bezeichnet. Sie generiert mit Hilfe des so genannten *K-Kombinators* (vgl. Programm 1.1 in Abschnitt 1.2.1) ein Kommando, das als beobachtbaren Wert gerade a besitzt und den internen Zustand unverändert lässt.

Die Operation „ & “ ist hier intuitiv zugänglicher als *flatten*; deshalb diskutieren wir sie zuerst. Sie liefert letztlich die sequenzielle Ausführung von Kommados. In dem Fragment

$$\ldots \mathit{readInt}(\mathit{file}) \;\&\; \lambda x \bullet \mathit{println}(x) \ldots$$

liest das Kommando *readInt* von der Datei (was den internen Zustand der Datei ändert); der beobachtbare Wert ist die gelesene Zahl, die an die Folgefunktion übergeben (und dort an x gebunden) wird.

Durch die Verwendung des so genannten *S-Kombinators* wird die Definition zwar ungemein elegant, aber nicht unbedingt verständlicher. Deshalb betrachten wir die Definition in langer Form. Sei s ein Zustand und *cmd* ein Kommando; dann gilt:

$$\begin{array}{lll} (\mathit{cmd} \,\&\, f)(s) = & \text{LET} & (a,\, s') \;\; = (\mathit{cmd}.\mathit{observe}(s),\; \mathit{cmd}.\mathit{evolve}(s)) \\ & & \mathit{cmd}' \;\; = f(a) \\ & & (b,\, s'') \;\; = (\mathit{cmd}'.\mathit{observe}(s'),\; \mathit{cmd}'.\mathit{evolve}(s')) \\ & \text{IN} & \\ & & (b, s'') \end{array}$$

(Man sieht leicht, dass die Kurzform des Programms aus dieser Langform durch schlichtes Einsetzen entsteht.) Mit dem Kommando *cmd* wird zunächst der sichtbare Wert a aus dem Zustand s extrahiert und der Zustand in den Folgezustand s' transformiert. Die Funktion f berechnet dann aus a das nächste Kommando, das auch gleich auf s' angewandt wird und so den nächsten Wert b und den weiteren Folgezustand s'' liefert.

Wir geben auch eine Variante der Operation „ & “ an, bei der das zweite Argument direkt als Monade gegeben ist und nicht erst berechnet werden muss (was sich sich gut mit dem K-Kombinator aus Programm 1.1 beschreiben lässt). Dies ist gerade bei der wichtigen Ein-/Ausgabe-Monade eine häufige Situation. Hier haben wir oft Ausdrücke der Bauart

$$\ldots \,\&\, \mathit{write}(x) \,\&\, \mathit{write}(y) \,\&\, \mathit{write}(z) \,\&\, \ldots$$

Die Operation *flatten* sieht bei der Zustands-Monade endgültig mystisch aus, wird aber durch eine etwas längere Schreibung deutlicher.

$$\mathit{flatten}\ \mathit{cc} = \mathit{cc} \,\&\, (\lambda\, c \bullet c)$$

Hier wird zunächst *cc* auf den gegebenen Zustand s angewandt. Das Ergebnis ist ein neues Kommando c und ein Folgezustand s'. Auf dieses Kommando c wird jetzt die Funktion *id* angewandt, was c erhält. Entsprechend der Definition von „ & “ wird dieses c auf s' angewandt. Mit anderen Worten, *flatten cc* wendet das von *cc* generierte Kommando sofort wieder an.

Anmerkung 1: Diese Definition von Machine entspricht ziemlich genau den so genannten Moore-Automaten aus der Theoretischen Informatik. Diese werden ebenfalls über zwei Funktionen charakterisiert, einen internen Zustandsübergang und eine externe Ausgabefunktion.

Anmerkung 2: In der Literatur wird eine leicht abweichende Definition für State benutzt. An Stelle unserer beiden Funktionen observe und evolve gibt es nur eine einzige Funktion mit der Funktionalität $(State \to \alpha \times State)$*. Das ist isomorph zu unserer Darstellung, aber in der Implementierung etwas effizienter. Trotzdem ziehen wir die Trennung in separate Funktionen vor, weil dies konzeptuell klarer ist.*

Anmerkung 3: Es ist kein Zufall, dass auch ein weiterer vielbeachteter Ansatz zur Integration von imperativer Programmierung in die funktionale Welt den Maschinenbegriff bemüht. Die so genannten Abstract State Machines von Yuri Gurevich (die ursprünglich Evolving Algebras hießen) realisieren in einem syntaktisch anderen, aber konzeptuell verwandten Ansatz die Idee von Automaten als mit der Zeit änderbare Funktionen [65, 64, 82].

11.2.4 Spezielle Zustands-Monaden

Weil *Machine* eine generische Struktur ist, können wir sie mit verschiedenen Typen für *State* instanziieren und erhalten so eine Reihe konkreter Zustands-Monaden. Einige kleine Beispiele wollen wir im Folgenden betrachten. Eine besonders interessantere Verwendung der Zustands-Monade wird sich später (in Kapitel 13) ergeben, wenn wir Optimierungstechniken für den Heap-Speicher diskutieren. Aber die wichtigsten Anwendungen werden wir erst in den Kapiteln 17 bis 20 kennen lernen, wenn wir Ein-/Ausgabe für funktionale Programme erörtern.

Programm 11.6 liefert bestenfalls eine Art von Skelett für das Arbeiten mit Zustands-Monaden. Denn es werden im Wesentlichen nur Operatoren zur Komposition von monadische Kommandos bereitgestellt. Was noch fehlt, um konkrete Zustands-Monaden zu kreieren, sind vier Dinge:

1. Elementare *Basisoperationen*, aus denen sich umfassendere Kommandos zusammensetzen lassen.
2. Die *Einbettung* der monadischen Operationen in umgebende rein funktionale Ausdrücke.
3. Eine *menschlichere Notation*, die nicht zu monströsen λ-Ausdrücken führt.
4. *Kompositionstechniken*, mit denen sich Monaden über verschiedenen Zustandsarten zusammenfügen lassen.

(1) Basisoperationen

Die Kompositionsoperatoren aus Programm 11.6 sind universell für alle Arten von Zustands-Monaden; sie stellen also eine Art generischer Rahmen-Automat dar. Daraus lassen sich spezifische Automaten machen, indem der Parameter für den verborgenen Zustand mit einem konkreten Typ instanziiert wird und

über diesem geeignete Basisoperatoren bereit gestellt werden. So gibt es zum Beispiel bei denEin-/Ausgabe-Monaden, die für die Interaktion mit dem Betriebssystem zuständig sind, elementare Kommandos wie *openFile*, *readFile*, *readString*, *deleteDirectory* usw. Diese elementaren Kommandos machen die eigentliche Substanz jeder konkreten Zustands-Monade aus. Damit sieht eine typische Definition folgendermaßen aus:

```
STRUCTURE MySpecialMonad = {
  EXTEND Machine(MyState) RENAMING Com AS MyMonad
  ...
  «spezifische Operationen»
  ...
}
```

(2) Einbettung in rein funktionale Ausdrücke

Wenn Monaden in funktionalen Programmen verwendet werden sollen, dann braucht man Operationen zur Einbettung in die funktionale Umgebung. Diese Operatoren haben typischerweise folgende Form:

```
STRUCTURE MySpecialMonad = {
  EXTEND Machine(MyState) RENAMING Com AS MyMonad
  ...
  FUN new: MyState → MyMonad Void
  FUN exec: MyMonad α → α
  ...
}
```

Damit kann die Einbettung dann folgendermaßen erfolgen:

$$\ldots\ f(exec(new(initial)\ \&\ \ldots\ \&\ \ yield(a)))\ \ldots$$

Der Ausdruck *initial*: *MyState* liefert den Anfangszustand, mit dem die monadische Berechnung startet. Der letzte monadische Wert – hier a – ist der Wert des gesamten Ausdrucks *exec*(...).

Anmerkung: Diese Einbettung sieht bei der wichtigsten Monade – der Ein-/Ausgabe-Monade – anders aus! Denn hier ist der Anfangszustand der Zustand des Rechners beim Programmstart. Konsequenterweise kann ein Term der Ein-/Ausgabe-Monade auch nicht *in funktionale Ausdrücke eingebaut werden. (Mehr dazu in Kapitel 17.)*

Das ist übrigens eine der hässlicheren Eigenschaften der Ein-/Ausgabe-Monade: Sobald irgendwo eine ihrer Operationen benutzt wird, muss auch alles „darum herum" monadisch sein. Diese „infektiöse" Natur der Ein-/Ausgabe-Monade macht sie in den Augen vieler Leute nur bedingt tauglich für elegante Programmierung.

(3) Notation

Notationelle Variationen, mit denen monadische Ausdrücke leserlich gemacht werden sollen, werden wir am Ende dieses Kapitels (in Abschnitt 11.3) vorstellen.

(4) Kompositionalität

Ein besonders schwieriges Thema ist die *Komposition* von Monaden. Diese Schwierigkeit gilt generell für die Monadentheorie. Aber wenn wir den theoretischen Purismus hintan stellen, dann können wir für die praktisch relevanten Fälle durchaus brauchbare Lösungen finden.

Das zentrale Problem liegt darin, dass der fundamentale Kompositionsoperator „&: $M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta$“ nur für Monaden über der gleichen Art von *State* definiert ist. Das führt dazu, dass man alle monadischen Aktivitäten unter einen einzigen Supermonaden packt – z. B. in OPAL die *Com*-Monade, in anderen Sprachen die *IO*-Monade. Damit ist man aber letztlich genau bei dem Problem gelandet, für das die imperativen Sprachen zurecht kritisiert werden: Wenn alles – von einfachen Variablen bis hin zu Dateisystem und Netzwerkumgebung – in einem einzigen Megazustand zusammengefasst wird, dann sind alle Modularisierungsregeln des Software-Engineering a priori verletzt.

Hier kann man – pragmatisch – Abhilfe schaffen, indem man den Operator „ & “ allgemeiner definiert. Nehmen wir an, wir haben zwei konkrete Zustands-Monaden:

```
STRUCTURE Monad1 = { EXTEND Machine(S1) RENAMING Com AS M1
                ...
}
STRUCTURE Monad2 = { EXTEND Machine(S2) RENAMING Com AS M2
                ...
}
```

Betrachten wir jetzt die Komposition zweier monadischer Kommandos aus den beiden unterschiedlichen Maschinen:

```
... m1: M1 α  &  m2: M2 β ...    -- Typfehler
```

Dieser Ausdruck ist falsch; ein Operator „ & “ dieses Typs existiert nicht. Aber in der Praxis ist diese Situation gang und gäbe. Also müssen wir aus pragmatischen Gründen das Konzept der Zustands-Monaden entsprechend erweitern.

Festlegung (Komposition von Zustands-Monaden)

Wenn zwei monadische Kommandos der obigen Art mit „ & “ komponiert werden, dann wir implizit eine weitere monadische Struktur der folgenden Bauart generiert:

```
STRUCTURE M1×2 = {
  EXTEND Machine(S1 × S2) RENAMING Com AS M1×2
                                        ...
}
```

Die monadischen Operationen beider Maschinen werden implizit nach $M_{1\times 2}$ gecastet. Die einzelnen Basisoperationen werden so erweitert, dass sie auf dem jeweils anderen Teilzustand die Identität sind.

11.2.5 Zähler als Zustands-Monaden

Eine der einfachsten Instanzen von *Machine* entsteht, wenn man den Parameter *State* mit *Int* instanziiert. Dann sind die Zustände einfach nur Zahlen. Dies ist in Programm 11.7 beschrieben. Man beachte, dass hier sowohl *observe* als auch *evolve* den Typ $Int \to Int$ haben.

Programm 11.7 Die *Counter*-Monade

```
STRUCTURE Counter = {
  EXTEND Machine(Int) RENAMING Com AS Counter

  FUN newCounter: Int → Counter Void                  -- kreiere Counter
  DEF newCounter(i) = (K nil, K i)

  FUN tick: Counter Void                              -- hochzählen
  DEF tick = (K nil, _ + 1)

  FUN current: Counter Int                            -- Zählerstand liefern
  DEF current = (id, id)

  FUN exec: Counter α → α                             -- in Umgebung einbetten
  DEF exec c = (c.observe)(0)
}
```

- Mit *newCounter* wird eine Zähler-Monade mit einem internen Startwert erzeugt.
- Mit *tick* wird der Zähler weitergeschaltet.
- Mit *current* kann man den Zählerstand abfragen.
- Mit *exec* lässt sich die Zähler-Monade in eine umfassende Funktion einbetten. Wenn *current* die letzte Aktion ist, dann wird der letzte Zählerstand als Wert abgeliefert, ansonsten *nil*. Man beachte, dass bei $c.observe$ das Argument 0 beliebig ist; jeder andere Zustand liefert das gleiche Resultat, weil *newCounter* ohnehin einen Startzustand setzt.

Typische Programmfragmente mit der Zähler-Monade haben folgende Form:

$$\ldots f(\ exec(\ newCounter(0)\ \&\ \ldots\ \&\ tick\ \&\ \ldots\ \&\ current\ \&\ \lambda x \bullet\ \ldots\ )\ )\ldots$$

11.2.6 Generatoren als Zustands-Monaden

Häufig muss man in einem Algorithmus neue Symbole generieren, die alle voneinander verschieden sein sollen. Dies lässt sich mit Hilfe einer Maschinen-Monade realisieren, die eine Folge von Zahlen liefert. Letztlich ist diese Monade eine Variante der Zähler-Monade, bei der die Operationen *tick* und *current* zur Operation *next* verschmolzen sind.

Programm 11.8 Die *Counter*-Monade

```
STRUCTURE Generator = {
  EXTEND Counter
    RENAMING (Counter, newCounter) AS (Generator, newGenerator)

  PRIVATE tick, current

  FUN next: Generator Int                          -- implizit hochzählen
  DEF next = tick & current
}
```

Man sieht sofort, dass man die Definition von *next* auch direkt hätte hinschreiben können:

```
DEF next = (id, _ + 1)
```

Aber in der Form von Programm 11.8 ist der Dokumentationswert wesentlich höher.

11.2.7 Ein-/Ausgabe als Zustands-Monade

Die wichtigste Anwendung der Zustands-Monaden ist die so genannte *Ein-/Ausgabe-Monade*, also die Behandlung von Ein-/Ausgabe in funktionalen Sprachen. Hier ist der Parameter *State* nicht direkt angebbar, sondern nur im Compiler intern verfügbar. Denn de facto handelt es sich dabei um den „Maschinenzustand“, also die Gesamtheit der Plattendateien, der Terminals, der externen Geräte usw. – bis hin zum Internet. Dieser interne Zustand wird praktisch vom Laufzeitsystem über Betriebssystem-Zugriffe realisiert. Wir gehen in Kapitel 17 und 19 noch einmal näher auf diese Fragen ein.

11.3 Spezielle Notationen für Monaden

Die Basisnotation, die wir bisher für Monaden eingeführt haben, erzeugt in der Praxis hässliche Ungetüme der Bauart

$$\ldots\, askInt(\texttt{"}x = ?\texttt{"}) \,\&\, \lambda x \bullet (askInt(\texttt{"}y = ?\texttt{"}) \,\&\, \lambda y \bullet (write(x + y) \ldots\ ))$$

Dieser Stil – der z. B. in OPAL notwendigerweise benutzt werden muss – wird schnell unerfreulich, wenn man es mit größeren Ein-/Ausgabe-Programmen zu tun hat. Das kann mit ein bisschen *Syntactic sugar* in eine wesentlich leserlichere Form gebracht werden.

11.3.1 Die Operatoren „→“ und „;“

Der λ-Operator ist eines der wenigen Sprachkonstrukte, mit denen neue Namen erzeugt und (bei der Applikation) an Werte gebunden werden können. Es bietet sich an, für die Kombination aus diesem Bindungsoperator und seiner unmittelbaren Applikation eine spezielle Notation einzuführen (ähnlich wie auch LET und WHERE spezielle Notationen für die direkte Applikation von λ-Ausdrücken sind). Damit lässt sich der obige Ausdruck sofort leserlicher schreiben:

$$\ldots \mathit{askInt}(\texttt{"}x = ?\texttt{"}) \to x\,;\ \mathit{askInt}(\texttt{"}y = ?\texttt{"}) \to y\,;\ \mathit{write}(x + y)\ldots$$

Je nach Geschmack und Anwendung kann man auch vorziehen, den Pfeil in der anderen Richtung zu haben (so wie auch die Verwendung von LET und WHERE oft Geschmackssache ist):

$$\ldots x \leftarrow \mathit{askInt}(\texttt{"}x = ?\texttt{"})\,;\ y \leftarrow \mathit{askInt}(\texttt{"}y = ?\texttt{"})\,;\ \mathit{write}(x + y)\ldots$$

Das sieht (nicht zufällig) einer Folge von Zuweisungen in imperativen Sprachen wie PASCAL sehr ähnlich. Es gibt aber fundamentale Unterschiede:

- Das Mixfixsymbol „$\_ \to \_\,;\,\_$“ ist ein *typisierter* Operator und bewirkt deshalb eine wesentlich höhere Typsicherheit des Gesamtprogramms.
- Der Operator „$\_ \to \_\,;\,\_$“ ist ein Derivat von „&“, und der Operator „&“ ist selbstdefiniert. Deshalb kann man ihn in unterschiedlichen Variationen definieren. Solche Variationen können z. B. automatisches Exceptionhandling vorsehen. (Genaueres werden wir in Kapitel 17 sehen.)

Festlegung (Der „→“-Operator)

Wir verwenden die Notationen

$m \to x\,;\ldots$ —— *m ist ein monadischer Ausdruck*

bzw.

$x \leftarrow m\,;\ldots$ —— *m ist ein monadischer Ausdruck*

als notationelle Varianten der Basisnotation

$m\ \&\ \lambda x \bullet \ldots$

Damit ist diese Notation keine Abweichung vom rein funktionalen Paradigma.

11.3.2 Erweitertes LET

In der Sprache HASKELL wird die →-Notation mit dem Schlüsselwort *do* eingeleitet. Wir verwenden hier eine Verallgemeinerung, die interesanter ist.

Im obigen Beispiel ist es irrelevant, in welcher Reihenfolge die beiden Werte x und y beschafft werden. In der Sprache OPAL ist für solche Situationen das LET-Konstrukt verfügbar. Damit könnte man den obigen Term folgendermaße schreiben:

```
... LET x ← askInt("x = ?")
        y ← askInt("y = ?")
    IN  write(x + y)
```

Jetzt wählt der Compiler eine beliebige Reihenfolge aus (die natürlich mit eventuell vorhandenen kausalen Abhängigkeiten verträglich sein muss).

Man sollte sich aber klar machen, dass das wesentlich von der Verwendung des Gleichheitszeichens verschieden ist. Das sieht man an folgender etwas aufgeblähten Variante des obigen Terms:

```
... LET cx = askInt("x = ?")   -- Typ cx: Com Int
        x ← cx                 -- Typ x: Int
        cy = askInt("y = ?")   -- Typ cy: Com Int
        y ← cy                 -- Typ y: Int
    IN  write(x + y)
```

Hier sind *cx* und *cy* die *monadischen Kommandos* selbst, während x und y die Werte sind, die bei der *Ausführung* der Kommandos abgeliefert werden.

11.3.3 Monaden-Casting

Mit dem erweiterten LET-Konstrukt hat man schon ein wesentliches Stück an Eleganz zurückgewonnen. Aber es bleiben noch Wünsche offen. Um dies zu sehen, betrachten ein Fragment eines Beispiels, das uns in Kapitel 17 noch beschäftigen wird. Nehmen wir an, wir haben einen Typ für arithmetische Ausdrücke in der Form von abstrakten Syntaxbäumen (vgl. Abschnitt 3.1):

```
TYPE Tree = add(left: Tree, right: Tree)
          | ...
```

Nehmen wir weiter an, dass bei der Auswertung Interaktionen mit dem Nutzer stattfinden müssen. Also ist die zugehörige Operation *eval* monadisch:

```
FUN eval: Tree → IO Int
DEF eval(add(l, r)) = LET x ← eval(l)
                          y ← eval(r) IN yield(x + y)
    ...
```

Was hier besonders stört, ist der Verlust der schönen Rekursionsstruktur. Denn eigentlich möchte man solche Funktionen in folgendem Stil schreiben:

```
DEF eval(add(l, r)) = eval(l) + eval(r)   -- Wunschdenken
```

Diese Form reflektiert direkt den Aufbau der Daten. In der LET-Klausel geht dies jedoch völlig unter. Retten lässt sich die schöne Struktur nur, indem man eine „monadische Addition“ einführt (mittels Overloading):

$$\begin{array}{l}\textsc{fun}\ \_ + \_\colon\ IO\ Int \times IO\ Int \to IO\ Int\\ \textsc{def}\ a + b = \textsc{let}\ x \leftarrow a\\ \qquad\qquad\qquad\quad y \leftarrow b\ \textsc{in}\ yield(x + y)\end{array}$$

Dieser individuelle Ansatz ist natürlich keine Lösung des generellen Problems, denn er verlangt vom Programmierer, für jede benötigte Operation ihr monadisches Gegenstück zu definieren. (Paul Hudak zeigt in [81], wie sich mit Hilfe von geeigneten polymorphen Lifting-Operationen entsprechende Instanzen der Typklassen *Num*, *Fractional* und *Floating* programmieren lassen; dabei sieht man, dass der Aufwand lästig, aber beherrschbar ist.)

Aber das Lifting zu monadischen Formen geschieht auf so mechanische Weise, dass sich eine Automatisierung anbietet: Eine moderne Programmiersprache sollte ***automatisches Monadenlifting*** vorsehen: Zu jeder Funktion der Art

$$\textsc{fun}\ f\colon\ \alpha \to \beta$$

kann man entsprechende Gegenstücke

$$\begin{array}{l}\textsc{fun}\ \widetilde{f}\colon\ \alpha \to IO\ \beta\\ \textsc{fun}\ \overline{f}\colon\ IO\ \alpha \to IO\ \beta\end{array}$$

definieren. (Letztere ist ohnehin als $f *$ generell verfügbar.) Das funktioniert entsprechend auch bei mehreren Argumenten und Resultaten. Dieses Lifting wird dann bei Bedarf als Typanpassung (vgl. Abschnitt 7.1.2) automatisch vom Compiler vorgenommen.

Voraussetzung für dieses Vorgehen ist allerdings, dass Monaden als spezielles Sprachfeature behandelt werden und somit dem Compiler bekannt sind. In Sprachen wie z. B. HASKELL sind Monaden aber als benutzerdefinierbar vorgesehen.

12

Netter Stack und böse Queue

Wie anfangs man geirrt, das findet man am Ende.
Oh, dass ich wenigstens auf halbem Wege fände!
Rückert (Weisheit des Brahmanen)

Neben einigen wenigen vordefinierten Datenstrukturen (meist Listen) bieten die funktionalen Sprachen ein wesentliches Feature zur Definition eigener Datenstrukturen an: *rekursive Typdefinitionen* wie z. B.

```
TYPE Tree α = tree ( left = Tree α, right = Tree α)   -- inner node
            | ⟨_⟩ ( val = α)                          -- leaf
            | Empty                                   -- empty
```

Aber auch verschränkt rekursive Definitionen sind möglich:

```
TYPE Tree α = tree ( node = α, subtrees = Forest α)   -- inner node
            | Empty                                   -- empty
TYPE Forest α = List (Tree α)                         -- subtrees
```

Allerdings gibt es dabei teilweise gravierende Effizienzprobleme. Im Folgenden betrachten wir diese Probleme phänotypisch am Fall der *listenartigen Strukturen*, nicht nur, weil diese am überschaubarsten sind, sondern auch, weil Listen in der Funktionalen Programmierung eine dominante Rolle spielen.

Dabei werden wir sehen, dass nur beim Stack die Operationen (wegen ihrer Strukturverträglichkeit) in konstanter Zeit $\mathcal{O}(1)$ ausführbar sind. Die anderen Operationen, also z. B. *append* „am falschen Ende“, Konkatenation, Längenberechnung etc. brauchen meistens lineare Zeit $\mathcal{O}(n)$ – zumindest dann, wenn man sie naiv implementiert. Deshalb werden wir ausgefeiltere Realisierungen studieren, die einen „amortisierten Aufwand“ $\mathcal{O}(1)$ haben, allerdings immer noch mit einem höheren konstanten Faktor als die Stack-Operationen. Das Prinzip dieser Effizienzverbesserungen ist dabei immer das gleiche:

Man verzögere alle „teuren“ Operationen so lange,
bis ihre Ausführung unumgänglich ist.

Das ist im Grunde das gleiche Prinzip, das wir schon bei den *lazy* Strukturen in Kapitel 2 angewendet haben. Dort ging es darum, die unnötige Ausführung unendlich langer Prozesse zu verhindern; jetzt wollen wir nur endliche, aber teure Prozesse verzögern, bis sie billiger werden. Trotzdem wird uns das Prinzip der Laziness auch hier gute Dienste leisten. Deshalb brauchen wir für den Rest dieses Kapitels die Listen in beiden Varianten.

Wenn wir von Prinzipien des Software-Engineering ausgehen, dann sollten wir die Strukturen dieses Kapitels in einem Package zusammenfassen. Dieses Package ist in Programm 12.1 skizziert. Die Spezifikationen *LIFO* und *FIFO*

Programm 12.1 Das Package für die Sequenz-artigen Strukturen

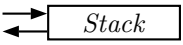
```
PACKAGE Sequences = {
  SPECIFICATION LIFO[α: Type] = {...}          -- last-in first-out
  SPECIFICATION FIFO[α: Type] = {...}          -- first-in first-out
  STRUCTURE Stack: LIFO = {...}                -- to be defined later
  STRUCTURE Queue: FIFO = {...}                -- to be defined later
  PRIVATE STRUCTURE Lists[α: Type] = {
    TYPE List = _ ·: _ (ft = α, rt = List)     -- prepend
              | Empty                          -- empty
    FUN |_|: List → Nat                        -- Länge der Liste
    FUN _ ++ _ : List × List → List            -- Konkatenation
    FUN revert: List → List                    -- Invertierung der Liste
  }
  PRIVATE STRUCTURE LazyLists[α: Type] = {
    TYPE List = _ ·: _ (ft = α, rt = LAZY List)  -- prepend
              | Empty                          -- empty
    FUN |_|: List → Nat                        -- Länge der Liste
    FUN _ ++ _ : List × List → List            -- Konkatenation
    FUN revert: List → List                    -- Invertierung der Liste
  }
}
```

sowie ihre Implementierungen *Stack* und *Queue* sind der Gegenstand dieses Kapitels. Zur Realisierung verwenden wir sowohl normale Listen als auch lazy Listen; da es sich um Hilfsstrukturen handelt, verstecken wir sie nach außen.

Man beachte, dass wir den Namen *List* sowohl für den Typ der normalen Listen als auch für den Typ der lazy Listen verwenden. Der Konflikt wird über den Strukturnamen aufgelöst. Außerdem verwenden wir hier keine polymorphen Typen, sondern machen die ganze Struktur parametrisch.

Wir benötigen ein paar Standardoperationen wie z. B. die Länge einer Liste, die Konkatenation zweier Listen oder die Invertierung einer Liste. Wegen ihrer Einfachheit verzichten wir aber darauf, diese Operationen hier explizit zu programmieren.

12.1 Wenn Listen nur anders heißen: Stack



Die in der Einleitung dieses Kapitels über einen rekursiven Typ eingeführten Listen haben gerade ein *Stack-artiges* Verhalten: Hinzufügen und Wegnehmen finden nur am gleichen Ende statt (*last-in first-out*, *LIFO*). Dies ist in Programm 12.2 definiert, in dem einige Komponenten des Packages *Sequences* aus Programm 12.1 im Detail ausformuliert sind.

Programm 12.2 Die Struktur *Stack*

```
SPECIFICATION LIFO[α: Type] = {
  TYPE Stack                              -- Stacks sind LIFO-Listen
  FUN ◊: Stack                            -- leerer Stack
  FUN ⟨_⟩: α → Stack                      -- einelementiger Stack
  FUN _ ·: _: α × Stack → Stack           -- add, push, prepend
  FUN ft: Stack → α                       -- first, get, top
  FUN rt: Stack → Stack                   -- rest, remove, pop
  PROP ft(a ·: s) = a                     -- get ist last-in first-out
  PROP rt(a ·: s) = s                     -- remove ist last-in first-out
}

STRUCTURE Stack[α] : LIFO[α] = {
  USE Lists[α]                            -- verwende die Struktur der Listen
  PRIVATE TYPE Stack = List α             -- Stack = List (eager)
  PRIVATE DEF ⟨a⟩ = a ·: ◊
}
```

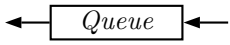
Die Spezifikation *LIFO* führt alle Stack-Funktionen ein und definiert die zwei wesentlichen Propertys, die das *Last-in first-out*-Verhalten ausmachen. Als Komfort haben wir noch den einelementigen Stack hinzugefügt.

Die Typisierung *Stack*: *LIFO* macht die gesamte Signatur, die in *LIFO* spezifiziert ist, für die Struktur *Stack* verfügbar. In den Kommentaren sind die häufigsten Namen angegeben, unter denen diese Operationen in der Literatur (und in Programmbibliotheken) geführt werden.

Man sieht, dass die Operationen der Struktur *Stack* im Wesentlichen die induzierten Operationen des Typs *List* sind. (Zur Erinnerung: Durch die Verwendung von USE *Lists* sind der Typ *List* und seine Operationen direkt verfügbar.) Die Einführung einer solchen Struktur erfüllt daher höchstens den Zweck, den Begriff „Stack“ – in Analogie zu anderen Programmiersprachen – auch in der Bibliothek zu haben.

Die Stack-Operationen sind wegen ihrer Strukturverträglichkeit mit der rekursiven Typdefinition *List* alle in konstanter Zeit $\mathcal{O}(1)$ ausführbar.

12.2 Wenn Listen zum Problem werden: Queue



Bei *Queue-artigem* Verhalten finden das Anfügen und das Wegnehmen an den entgegengesetzten Enden der Liste statt (*first-in first-out*, *FIFO*) und damit entgegen dem „natürlichen" Listenaufbau. Wir drücken dies dadurch aus, dass wir für *push* die Operation „am anderen Ende", also *append*, verwenden; die übrigen Operationen bleiben die gleichen wie vorher, weshalb wir auch die gleichen Identifier beibehalten. Damit ergeben sich die Spezifikation und Struktur in Programm 12.3, womit zwei weitere Komponenten aus Programm 12.1 ausformuliert werden.

Programm 12.3 Die Struktur *Queue*

```
SPECIFICATION FIFO[α: Type] = {
  TYPE Queue                                  -- FIFO-Listen
  FUN ◊: Queue                                -- leere Queue
  FUN ⟨_⟩: α → Queue                          -- einelementige Queue
  FUN _ :· _: Queue × α → Queue               -- add, push, append
  FUN ft: Queue → α                           -- first, get, top
  FUN rt: Queue → Queue                       -- rest, remove, pop
  PROP ft(◊ :· a) = a
  PROP ft(q :· a) = ft(q)  IF q ≠ ◊           -- get ist first-in first-out
  PROP rt(◊ :· a) = ◊
  PROP rt(q :· a) = rt(q) :· a  IF q ≠ ◊      -- remove ist first-in first-out
}

STRUCTURE Queue[α] : FIFO[α] = {
  USE Lists[α]                                -- verwende die Struktur der Listen
  PRIVATE TYPE Queue = List                   -- Queue = List
  PRIVATE DEF ◊ :· a = a ·: ◊
  PRIVATE DEF (x ·: q) :· a = x ·: (q :· a)   -- push (append)
}
```

Wie man sieht, sind die meisten Operationen noch die vom Typ *List* induzierten Konstanten und Selektoren. Aber der Konstruktor $\cdot\!:$ (*prepend*) ist jetzt verborgen. Dafür wird als *push*-Operation die *append*-Funktion $:\!\cdot$ verfügbar gemacht.

Man sieht sofort, dass die *push*-Operation $:\!\cdot$ hier den Aufwand $\mathcal{O}(n)$ hat. Da Queues aber außerordentlich häufige und wichtige Datenstrukturen sind, kann diese Ineffizienz nicht toleriert werden. Deshalb untersuchen wir im Folgenden Wege zur Verbesserung des Aufwands. Wir orientieren uns dabei an der Darstellung von Okasaki [107], wo auch weitere Details und Variationen nachzulesen sind. Allerdings adaptieren wir die Präsentation an unseren Kontext.

12.2.1 Variante 1: Queue = Paar von Listen

Wir beginnen mit einer einfachen Idee (die später noch verbessert werden wird). Die Queue besteht aus zwei Teillisten, von denen die hintere in invertierter Form gespeichert ist (s. Abbildung 12.1). Die „falsche“ Operation *append* wird als normales *prepend* an der hinteren Teilliste ausgeführt, *ft* und *rt* dagegen ganz normal an der vorderen Teilliste.

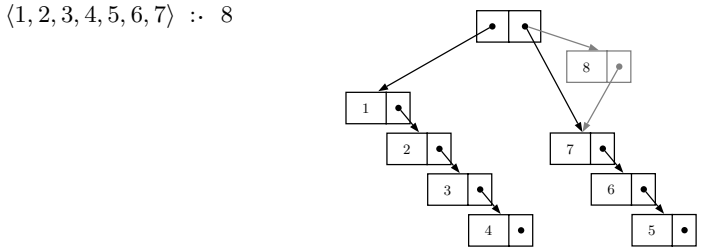


Abb. 12.1: Queue als Paar von Listen

Das einzige Problem entsteht, wenn die vordere Teilliste leer ist; spätestens dann muss *die hintere Liste revertiert* und als vordere genommen werden. Um spätere Optimierungen vorzubereiten werden wir diesen Umbau aber etwas früher vornehmen, nämlich bereits dann, wenn die vordere Liste kürzer würde als die hintere. Das heißt, wir garantieren folgende *Invariante*:

(I) *Die vordere Liste ist nie kürzer als die hintere.*

Diese Invariante wird in Programm 12.4 dadurch sichergestellt, dass bei den beiden Operationen, die Resultate vom Typ *Queue* erzeugen – nämlich :· und *rt* – jeweils eine Normalisierung mittels *norm* erfolgt.

Die neue Struktur *Queue* aus Programm 12.4 ist funktional völlig gleichwertig zu der alten Struktur *Queue* aus Programm 12.3. Im Package *Sequences* können also beide alternativ zueinander verwendet werden.

Wenn man Programm 12.4 betrachtet, sieht man sofort, dass die Operation *norm* tatsächlich die Invariante (I) garantiert. Und wegen dieser Invarianten brauchen die Operationen *ft* und *rt* nicht zu prüfen, ob die linke Liste leer ist. Denn sie kann nur leer sein, wenn auch die rechte Liste leer. Dann ist aber die ganze Queue leer und die beiden Operationen sind zu Recht undefiniert.

Natürlich sollte man das Programm noch optimieren, indem man die Längen der beiden Listen nicht ausrechnet (was ja wieder einen Aufwand $\mathcal{O}(n)$ bedeuten würde), sondern in dem Tupel mitspeichert:

TYPE *Queue* = (*List* α, *Nat*, *List* α, *Nat*) −− *Längen gespeichert*

Programm 12.4 Queue als Paar von Listen

```
STRUCTURE Queue[α] : FIFO[α] = {
  USE Lists[α]
  PRIVATE PART
    TYPE Queue = (List, List)                          -- Queue = Paar von Listen
    DEF ◊ = (◊, ◊)                                     -- leere Queue
    DEF (left, right) :· a = norm(left, a ·: right)    -- push (append)
    DEF ft(left, right) = ft(left)                     -- top
    DEF rt(left, right) = norm(rt(left), right)        -- pop
    FUN norm: Queue → Queue                            -- normalisieren
    DEF norm(left, right) = IF |left| ≥ |right| THEN (left, right)
                            IF |left| < |right| THEN (left ++ revert(right), ◊) FI
    PROP ∀(left, right): Queue • |left| ≥ |right|      -- Invariante (I)
}
```

Konstanter Aufwand (amortisiert)

Die Implementierung von Programm 12.4 hat tatsächlich konstanten Aufwand, genauer: einen amortisierten Aufwand $\mathcal{O}(1)$. Wir können hier diesen Begriff und die zugehörigen Beweistechniken nicht detailliert behandeln, sondern müssen auf [107] verweisen. Doch die Grundidee ist relativ einfach: Jede *prepend*-Operation auf der rechten Liste wird mit dem Aufwand 2 gezählt (statt mit dem korrekten Aufwand 1). Auf diese Weise sammelt man einen „Kredit“ an, der dann bei der *revert*-Operation wieder aufgebraucht wird.

Diese Argumentation zeigt aber auch, dass wir etwa einen Faktor 2 gegenüber der traditionellen imperativen Programmierung verlieren. Das ist der Preis, den man für die gewonnene Programmiersicherheit zahlen muss.[1]

Mit dem Design von Programm 12.4 haben wir noch ein Problem geschaffen: Die Verwendung der Konkatenation _ ++ _ braucht einen Aufwand in der Größenordnung der ersten Teilliste *left*. Dies kann man im Prinzip verbessern, indem man das Konkatenieren nicht wirklich durchführt, sondern stattdessen eine Liste von Listen verwaltet. (Darauf kommen wir in Abschnitt 12.3 in Abbildung 12.3 noch einmal zurück.) Wir verzichten darauf, diese Variante auszuprogrammieren, sondern verweisen wieder auf [107]. Denn der Zeitverlust durch die Konkatenation ist nicht unser schlimmstes Problem.

Schwerer wiegt die Tatsache, dass die Dauer der *Queue*-Operationen nicht vorhersagbar ist. Meistens macht die Anwendung von *rt* oder :· den Aufwand $\mathcal{O}(1)$, aber wenn die kritische Länge erreicht ist, dann ist der Aufwand plötzlich $\mathcal{O}(n)$, weil revertiert und konkateniert wird. Diese zufällige Zeitdau-

[1] Wenn man bedenkt, dass bei den modernen objektorientierten Sprachen JAVA und .NET der Verlangsamungsfaktor – je nach Compiler – in der Größenordnung $10 - 20$ liegen kann, dann ist unser Faktor 2 geradezu harmlos.

er ist in Realzeitanwendungen nicht tolerierbar. Dort ist Vorhersagbarkeit des *worst-case* Verhaltens wichtiger als Effizienz des *average-case* Verhaltens.[2]

Also sollten wir nach Lösungen suchen, die den Aufwand immer in der konstanten Größenordnung $\mathcal{O}(1)$ halten.

12.2.2 Variante 2: Faulheit macht kalkulierbar

Auf den ersten Blick mag es verblüffend sein, aber bei genauerem Nachdenken leuchtet es dann doch ein: Durch die Verwendung von *lazy Listen* kann man die Queues realzeitfähig[3] machen. Das Problem besteht nämlich darin, dass die Funktion *revert* zu hektisch ist und gleich die ganze rechte Liste umdreht, wodurch der Aufwand $\mathcal{O}(n)$ entsteht. Dabei reicht es, wenn sie die Elemente eins nach dem anderen herausrückt. Wenn aber immer nur ein Element verarbeitet wird, werden nur einige wenige Instruktionen ausgeführt und wir haben unseren gewünschten konstanten Aufwand $\mathcal{O}(1)$.

Anmerkung: Man beachte, dass die folgenden Überlegungen ***beide*** *Eigenschaften von Laziness benötigen: (1) Die Auswertung von Argumenten wird verzögert, bis sie tatsächlich gebraucht werden. (2) Argumente, die mehrfach vorkommen, werden trotzdem nur einmal ausgewertet (vgl. Abschnitt 1.3). Die Simulation von Laziness durch λ-Abstraktion (vgl. Kapitel 2) erfasst nur den ersten Aspekt; deshalb reicht sie für die folgenden Implementierungen nicht. Man braucht* ***echte Laziness****.*

Wir stellen Queues so dar, dass wir die vordere (linke) Liste lazy machen. Die hintere (rechte) Liste kann eine normale Liste bleiben. Das reicht allerdings nicht aus. Wir brauchen noch etwas, das Okasaki [107] einen *Schedule* nennt. Dieser Schedule legt fest, *wann* verzögerte Operationen ausgeführt werden, also wann UNQUOTE (vgl. Kapitel 2) ausgeführt wird. Das Hübsche an der Geschichte ist, dass man diesen Schedule ganz einfach über eine weitere lazy Liste realisieren kann (die auf der Implementierungsebene vom Compiler zu einem schlichten Pointer degradiert wird und damit ganz billig ist; dazu wird die oben erwähnte Eigenschaft (2) von lazy Listen gebraucht). Programm 12.5 enthält die Implementierung.

Die wichtigste Änderung gegenüber Programm 12.4 ist, dass Queues jetzt Tripel sind, bestehend aus der vorderen Liste (lazy), dem Schedule (lazy) und der hinteren Liste (eager). Im alten Programm 12.4 war die zeitkritische Operation in der Funktion *norm* der Ausdruck $left +\!\!+ revert(right)$. Diesen ersetzen wir jetzt durch eine Funktion *rotate* mit drei Parametern. Der Effekt dieser Operation lässt sich folgendermaßen spezifizieren:

[2] Benutzer sind zufriedener, wenn eine interaktive Anwendung immer 1 Sekunde Antwortzeit hat, als wenn sie 99 Mal schon nach $\frac{1}{4}$ Sekunde antwortet und beim hundertsten Mal 25 Sekunden braucht – obwohl Letzteres über alle Antworten hinweg doppelt so schnell ist.

[3] Wir meinen hier natürlich die so genannte *weiche Realzeit* wie sie z. B. in interaktiven Programmen benötigt wird. *Harte Realzeit*, die z. B. in Steuerprozessoren gebraucht wird, wo man im Milli- oder Mikrosekundenbereich reagieren muss, ist in der heutigen Funktionalen Programmierung (noch) kein Thema.

Programm 12.5 Realzeit-Queues

```
STRUCTURE Queue[α] : FIFO[α] = {
  USE Lists[α]                                              -- strikte Listen
  USE LazyLists[α]                                          -- lazy Listen
  PRIVATE PART
    TYPE Queue = (LazyLists.List, LazyLists.List, Lists.List)
    DEF ◊ = (◊, ◊, ◊)                                       -- leere Queue
    DEF (left, sched, right) :· a = norm(left, sched, a ·: right)   -- push (append)
    DEF ft(left, sched, right) = ft(left)                   -- top
    DEF rt(left, sched, right) = norm(rt(left), sched, right)       -- pop
    FUN norm: Queue → Queue                                 -- normalize
    DEF norm(left, ◊, right) = (l, l, ◊) WHERE l = rotate(left, ◊, right)
    DEF norm(left, sched, right) = (left, rt(sched), right)
    FUN rotate: Queue → LazyLists.List
    DEF rotate(◊, sched, right) = ft(right) ·: sched        -- s. Text!
    DEF rotate(left, sched, right) =
                  ft(left) ·: rotate(rt(left), ft(right) ·: sched, rt(right))
}
```

```
  PROP rotate(left, sched, right) = left ++ reverse(right) ++ sched
  PROP rotate(left, ◊, right) = left ++ reverse(right)      -- Folgerung
```

Der zusätzliche Parameter *sched* ist ein so genannter *akkumulierender Parameter*, der die partiellen Resultate der Invertierung von *right* aufsammelt. Er ist anfangs leer.

Für die Korrektheit dieser Implementierung und für die Realzeitfähigkeit ist wichtig, dass der Datentyp *Queue* folgende *Invariante* (I′) einhält:

(I′) *Für das Tripel* $(left, sched, right)$ *gilt:* $|\, left| = |sched\,| + |\, right\,|$.

Damit wird automatisch auch die frühere Invariante (I) garantiert, denn aus der neuen Bedingung (I′) folgt insbesondere $|\, left| \geq |right\,|$.

Dass die Invariante (I′) gilt, sieht man relativ leicht. Für die leere Queue ist das trivialerweise der Fall. Wenn die Invariante verletzt wird, also bei den Operationen $:\cdot$ und *rt*, wird *norm* aufgerufen. Bei diesem Aufruf gilt $|\, left| = |sched\,| + |\, right\,| - 1$. Wenn *sched* nicht leer ist, führen wir $rt(sched)$ aus, wodurch (I′) wieder hergestellt wird. Wenn *sched* leer ist, liefert *norm* ein Tripel, in dem *left* und *sched* identisch sind und *right* leer ist; die Invariante gilt also ebenfalls wieder.

Die *Korrektheit* der Implementierung folgt aus der zweiten der oben angegebene Propertys: Wenn *sched* leer ist, ruft *norm* die Hilfsfunktion *rotate* auf und liefert damit den Wert $left +\!\!+ reverse(right)$ als linke Liste und macht die rechte Liste leer; ansonsten lässt *norm* die Listen *left* und *right* unverändert.

Bleibt also zu zeigen, dass die beiden *Propertys gelten*. Dazu genügt es, die erste zu zeigen. Man sieht sofort, dass für die beiden Parameter *sched* und

right der Wert $\mathit{reverse}(\mathit{right}) +\!\!+ \mathit{sched}$ über alle rekursiven Aufrufe hinweg unverändert bleibt, und dass der erste Parameter *left* im Laufe der rekursiven Aufrufe einfach nach außen kopiert wird. Wegen (I′) wissen wir außerdem, dass beim Aufruf von *rotate* die linke Liste um ein Element kürzer ist als die rechte Liste; deshalb ist im Abbruchfall *right* einelementig, und es reicht anstelle von $\mathit{reverse}(\mathit{right}) +\!\!+ \mathit{sched}$ einfach nur $\mathit{ft}(\mathit{right}) \cdot\!: \mathit{sched}$ zu schreiben.

Die Queue ist jetzt realzeitfähig

Nachdem wir uns vergewissert haben, dass die Implementierung von Programm 12.5 korrekt ist, bleibt die Frage der Realzeitfähigkeit. Während es bei den Korrektheitsbetrachtungen egal ist, ob die einzelnen Listen lazy sind oder nicht (sie haben ja gleiches funktionales Verhalten), muss man bei den Zeitbetrachtungen die Laziness mit einbeziehen. Um eine intuitive Vorstellung vom operationalen Ablauf zu bekommen, betrachten wir das Beispiel:

$$q\colon \mathit{Queue} = \mathit{rt}(\Diamond :\cdot 1 :\cdot 2 :\cdot 3 :\cdot 4) :\cdot 5$$

Das Ergebnis dieser Auswertung sollte also $q = \langle 2, 3, 4, 5\rangle$ sein.

In Abbildung 12.2 wird der Ablauf dieser Berechnung illustriert. Dabei zeigen wir immer das Tripel $(\mathit{left}, \mathit{sched}, \mathit{right})$ der drei Listen an, aus denen die Queue besteht. Einfache Kästen repräsentieren Elemente in normalen Listen, doppelt gestrichene Kästen repräsentieren die mit QUOTE gesperrten lazy Listen. In der mittleren Spalte ist jeweils die Zwischenform vor der Normalisierung gezeigt, in der rechten Spalte die endgültige Form.

- q_0 Wir beginnen mit der leeren Queue, die aus drei leeren Listen besteht.
- q_1 Beim Hinzufügen des ersten Elements (an die rechte Liste) muss die Normalisierung das Element nach links bringen. Man beachte, dass *left* und *sched* die gleiche (einelementige) Liste bezeichnen.
- q_2 Das zweite Element wird wieder rechts angefügt. Bei der Normalisierung wird die Liste *sched* wegen des Aufrufs $\mathit{rt}(\mathit{sched})$ in *norm* leer.
- q_3 Beim Hinzufügen des dritten Elements wird die rechte Liste zu lang. Jetzt führt *norm* auf *rotate*, was die gemeinsame Liste für *left* und *sched* erzeugt. Wegen der Laziness wird der rekursive Aufruf von *rotate* aber suspendiert.
- q_4 Beim Hinzufügen des vierten Elements wird wieder *norm* aufgerufen. Wegen des Aufrufs $\mathit{rt}(\mathit{sched})$ muss das suspendierte *rotate* einmal ausgeführt werden. (In diesem speziellen Fall gibt es keinen rekursiven Aufruf mehr; ansonsten würde er wieder suspendiert werden.) *sched* ist jetzt identisch mit der Restliste von *left*.
- q_5 Bei der Restbildung wird zunächst die Liste *left* verkürzt. Dadurch sind (in diesem Beispiel) *left* und *sched* wieder identisch. Die Normalisierung führt jetzt wieder $\mathit{rt}(\mathit{sched})$ aus.
- q_6 Beim Hinzufügen des fünften Elements bleibt dieses zunächst in der rechten Liste. Die Normalisierung führt wieder $\mathit{rt}(\mathit{sched})$ aus.

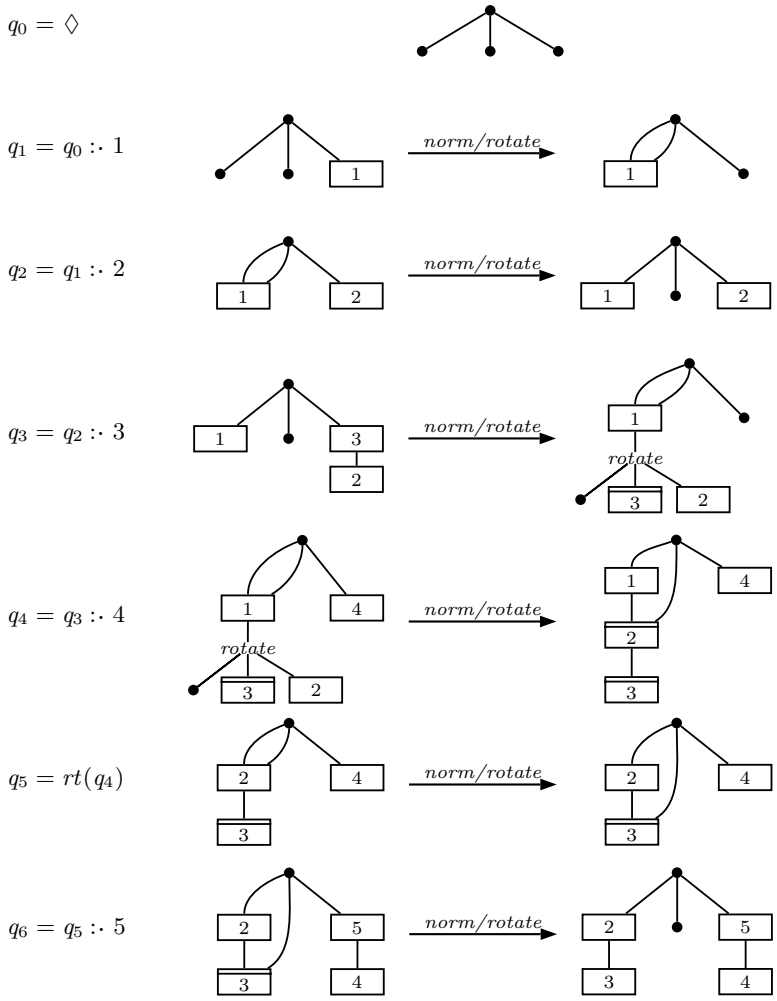


Abb. 12.2: Berechnungsablauf für $rt(\Diamond :\cdot 1 :\cdot 2 :\cdot 3 :\cdot 4) :\cdot 5$

Wie man sieht, repräsentiert die Liste *sched* denjenigen Teil der Liste *left*, der noch suspendiert ist, also noch auf die Auswertung wartet.

Man überzeugt sich schnell, dass jede *Queue*-Funktion nur einige wenige Operationen ausführt. Damit ist konstanter Aufwand $\mathcal{O}(1)$ garantiert.

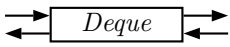
Interessant ist hier die Beobachtung, dass die *Korrektheit* der Implementierung ganz einfach zu sehen ist. Etwas kniffliger ist nur die Analyse der Effizienz. Dies ist eine relativ typische Situation für die Funktionale Programmierung.

12.3 Wenn Komplikationen sich häufen: Deque und Sequence

Trotz aller Komplikationen ist die Struktur *Queue* noch ziemlich harmlos. Kritischer wird es bei Anwendungen, die sowohl die Operationen von *Stack* als auch die von *Queue* brauchen; man spricht dann von *double-ended Queues* oder kurz von *Deques*. Und richtig kompliziert wird es, wenn dann noch die Konkatenation hinzukommt, also bei voll ausgebauten *Sequenzen*.

Wir können hier das Thema nicht in all seinen Facetten ausarbeiten, sondern müssen uns auf eine Skizze der wesentlichen Ideen beschränken. Wir verweisen deshalb auf das ausführliche Buch von Okasaki [107].

12.3.1 Double-ended Queues (*Deque*)



Die *double-ended Queue*, kurz *Deque*, vereinigt die Eigenschaften von Stack und Queue. Sie erlaubt also das Hinzufügen und Wegnehmen an beiden Enden. Allerdings wird dadurch eine Umbenennung einiger Operationen nötig. Damit ergibt sich die Signatur in Programm 12.6.

Programm 12.6 Die Spezifikation *DEQUE*

```
SPECIFICATION DEQUE[α: Type] = {
  TYPE Deque                                  -- FIFO+LIFO-Listen
  FUN ◇: Deque                                -- leere Deque
  FUN ⟨_⟩: α → Deque                          -- einelementige Deque
  FUN _ ·: _: α × Deque → Deque               -- addFirst, push, prepend
  FUN _ :· _: Deque × α → Deque               -- addLast, append
  FUN ft: Deque → α                           -- first, getFirst, top
  FUN lt: Deque → α                           -- last, getLast, bot
  FUN rt: Deque → Deque                       -- rest, removeFirst, pop
  FUN hd: Deque → Deque                       -- head, removeLast, front
}
```

Man hätte die Spezifikation auch mittels EXTEND und RENAMING direkt auf die Spezifikationen *FIFO* und *LIFO* abstützen können, aber das hätte die Lesbarkeit nicht unbedingt erhöht.

Die Grundidee der Implementierung ist in Abbildung 12.3 skizziert. Man geht von der Anfangsidee bei Queues aus, d. h. von zwei Listen, bei denen die hintere bei Bedarf invertiert wird (s. Abbildung 12.1). Um die Symmetrie zwischen vorne und hinten zu erhöhen, führt man die *revert*-Operationen jetzt allerdings nicht tatsächlich durch, sondern „merkt“ sie sich in einer Liste von Listen.

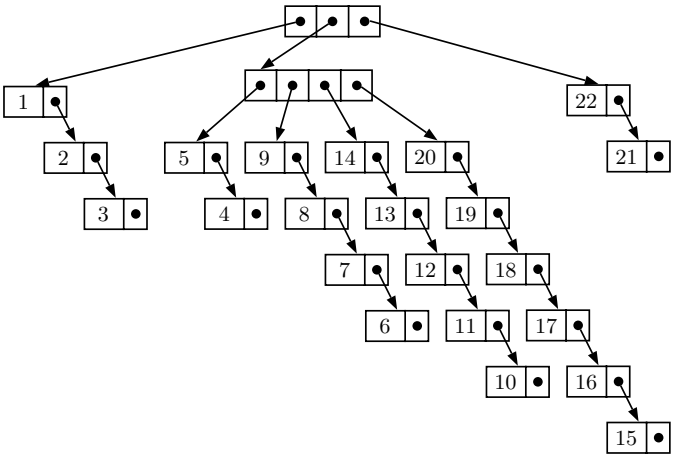


Abb. 12.3: Deque als Liste von Listen

- Die Invariante (I) wird so angepasst, dass *right* nur leer sein kann, wenn die innere Liste leer ist und *left* höchstens einelementig ist.
- Damit sind dann *lt* und *hd* ähnlich wie *ft* und *rt* programmierbar.

12.3.2 Sequenzen (*Catenable Lists*)

In vielen Anwendungen hat man neben den Stack-, Queue- oder Deque-Operationen vor allem noch die *Konkatenation* $S_1 +\!\!+ S_2$. Wir sprechen dann von *Sequenzen* oder *Catenable Lists* (s. Programm 12.7).

Programm 12.7 Die Spezifikation *Sequence*

```
SPECIFICATION Sequence[α: Type] = {
  TYPE Seq                                  -- Sequenzen
  FUN ◇: Seq                                -- leere Sequenz
  FUN ⟨_⟩: α → Seq                          -- einelementige Sequenz
  FUN _ ·: _: α × Seq → Seq                 -- addFirst, push, prepend
  FUN _ :· _: Seq × α → Seq                 -- addLast, append
  FUN ft: Seq → α                           -- first, getFirst, top
  FUN lt: Seq → α                           -- last, getLast, bot
  FUN rt: Seq → Seq                         -- rest, removeFirst, pop
  FUN hd: Seq → Seq                         -- head, removeLast, front
  FUN _ ++ _: Seq × Seq → Seq               -- Konkatenation
}
```

In einer naiven Implementierung hat die Konkatenation $S_1 +\!\!+ S_2$ einen Aufwand in der Größenordnung $\mid S_1 \mid$ des linken Arguments.

Variante 1: Sequenzen als Binärbäume

In einem ersten – ebenfalls recht naiven – Ansatz kann man die Konkatenation zum Konstruktor machen. Man würde dann folgende Typdefinition erhalten:

$$\begin{array}{lll} \text{TYPE } \mathit{Seq} = & \mathit{Empty} & -- \textit{ leere Sequenz} \\ & \langle \_ \rangle\,(\mathit{val}\colon \alpha) & -- \textit{ einelementige Sequenz} \\ & \_ +\!\!+ \_\,(\mathit{left}\colon \mathit{Seq},\ \mathit{right}\colon \mathit{Seq}) & -- \textit{ Konkatenation} \end{array}$$

Das Hinzufügen vorne und hinten lässt sich leicht realisieren:

$$\begin{array}{l} \text{DEF } a \cdot\colon S = \langle a \rangle +\!\!+ S \\ \text{DEF } S \colon\cdot\ a = S +\!\!+ \langle a \rangle \end{array}$$

Aber die Zugriffsoperationen *ft*, *rt*, *lt* und *hd* werden ziemlich ineffizient. (Wir verzichten darauf, ihre – recht offensichtlichen – Implementierungen anzugeben.) Eine gewisse Verbesserung kann man erreichen, indem man ausgefeiltere Baumimplementierungen nimmt wie z. B. *Rot-Schwarz-Bäume*.

Variante 2: Cat-Queues

Wenn wir nur die Zugriffsoperationen an einem Ende brauchen, also z. B. die Operationen *lt* und *hd* aus der Signatur streichen, dann gibt es auch noch eine einigermaßen effiziente Implementierung. Wir hatten bei Deques schon gesehen, dass wir die Ausführung der $+\!\!+$ -Operationen einfach verzögern können, indem wir die Sequenzen in einer Liste zwischenspeichern. Aufgrund ihrer Verwendung hat diese Liste das Verhalten einer Queue. Das führt auf folgenden Typ für „linkslastige“ Sequenzen. (Man beachte, dass wir den Typ *Queue* aus der entsprechend instanziierten Struktur $\mathit{Queue}[\mathit{Seq}]$ brauchen.)

$$\begin{array}{ll} \text{TYPE } \mathit{Seq} = & \mathit{Empty} \\ & \mathit{sq}(\mathit{ft} = \alpha,\ \mathit{susp} = \mathit{Queue}[\mathit{Seq}]\,.\,\mathit{Queue}) \end{array}$$

Damit ergeben sich dann z. B. die folgenden Definitionen für die zentralen Operationen (man beachte, dass *ft* schon ein Selektor ist):

$$\begin{array}{l} \text{DEF } \Diamond +\!\!+ \mathit{seq} = \mathit{seq} \\ \text{DEF } \mathit{seq} +\!\!+ \Diamond = \mathit{seq} \\ \text{DEF } \mathit{sq}(a,\ \mathit{queue}) +\!\!+ \mathit{seq} = \mathit{sq}(a, \mathit{queue} \colon\cdot\ \mathit{seq}) \\ \text{DEF } \mathit{rt}(\mathit{sq}(a,\ \Diamond)) = \Diamond \\ \text{DEF } \mathit{rt}(\mathit{sq}(a,\ \langle \mathit{seq} \rangle)) = \mathit{seq} \\ \text{DEF } \mathit{rt}(\mathit{sq}(a,\ \mathit{queue})) = +\!\!+ \,/\ \mathit{queue} \end{array}$$

(Dabei verwenden wir einen entsprechenden Reduce-Operator auf Queues.)

Man kann auch hier zeigen, dass der *amortisierte Aufwand* aller Operationen wieder $\mathcal{O}(1)$ ist.

Variante 3: Sequenzen

Die Implementierung der vorigen Version ist asymmetrisch und bedeutet daher große Kosten bei den „falschen“ Operationen *lt* und *hd*. Hier ist auch nur schwer Abhilfe möglich.

Man erkennt das Problem schnell, wenn man sich z. B. an Abbildung 12.1 orientiert, wo eine Queue als Paar von Listen realisiert ist. Wenn man auf dieser Basis eine „Konkatenation mit Verzögerung“ implementieren wollte, käme man auf folgende Situation (wobei der Querstrich andeutet, dass die Liste revertiert vorliegt):

$$(A, \bar{B}) +\!\!+ (X,\ \bar{Y}) = (A,\ \langle \bar{B},\ X\rangle,\ \bar{Y})$$

Allgemein erhalten wir in diesem Design eine Situation ähnlich der von Abbildung 12.3, nur mit dem Unterschied, dass die Listen in der mittleren Queue teilweise revertiert sind und teilweise nicht.

$$(A, BB, C) +\!\!+ (X, YY, Z) = (A,\ BB +\!\!+ C \cdot\!: X \cdot\!: YY,\ Z)$$

Zwar führt hier die Konkatenation der Sequenzen auf eine Konkatenation von Queues (oder Deques), aber wir können argumentieren, dass die Queue *BB* im Allgemeinen sehr kurz ist.

Fazit: Insgesamt zeigt sich aber, dass eine halbwegs effiziente Realisierung von Sequenzen kaum möglich ist, wenn wirklich alle Operationen gleichermaßen funktionieren sollen.

12.4 Arbeiten mit listenartigen Strukturen

Aus softwaretechnischer Sicht ist diese Fülle von ähnlichen Strukturen unangenehm, weil sie vom Programmierer verlangt, immer sehr detailliert über die benötigten Operationen und damit über die jeweils benötigte Listenvariante nachzudenken. Das sind aber rein implementierungstechnische Fragen. Und da die funktionale Programmierung den Anspruch erhebt, die Softwareentwickler von solchen Details zu befreien, muss man hier automatische Übersetzungen oder zumindest halbautomatische Hilfen anbieten. Das wird zwar im Moment von keinem Compiler tatsächlich gemacht, stellt aber keine grundsätzlichen Probleme dar:

- Eine Sprache wie OPAL hat im Prinzip die nötigen Features schon (im Ansatz) verfügbar. Wir können z. B. schreiben

  IMPORT *Sequence* α ONLY *Seq* $\Diamond$ $\cdot\!:$ *ft rt*

  Daraus kann der Compiler erkennen, dass hier nur die Struktur *Queue* gebraucht wird.
- Selbst wenn der Programmierer aus Bequemlichkeit einen kompletten Import verlangt,

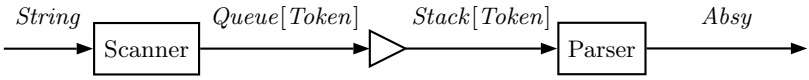
  IMPORT *Sequence* α COMPLETELY

kann der Compiler die Teilmenge der tatsächlich benötigten Operationen ausrechnen und die Implementierung entsprechend auswählen.

- In einer komfortableren Programmierumgebung wird man Notationen bereitstellen, die zu einer Signatur unterschiedliche Implementierungen auswählen lassen.

Neben diesen eher softwaretechnischen Maßnahmen muss allerdings noch etwas Weiteres geschehen: Wir brauchen *Konversionsroutinen* zwischen den einzelnen Strukturarten, weil man in verschiedenen Phasen eines Algorithmus oft verschiedene Listenarten hat.

Betrachten wir z. B. das Zusammenspiel von Scanner und Parser im Compilerbau. Dort haben wir eine Situation folgender Gestalt:



Man sieht aber bei den obigen Implementierungen unschwer, dass derartige Konversionsoperationen problemlos gestaltet werden können.

13

Compilertechniken für funktionale Datenstrukturen

... dass ein roter Faden durch das Ganze durchgeht.
Goethe (Die Wahlverwandtschaften)

Die im vorigen Abschnitt diskutierten Techniken bewegen sich alle im Rahmen der klassischen funktionalen Datenstrukturen. Wir haben keinerlei Konstrukte benutzt, die nicht in gängigen Sprachen verfügbar wären. Der Effekt ist, dass *die Datenstrukturen garantiert sicher sind*, allerdings immer noch *auf Kosten der Effizienz*. Denn wir haben zwar erheblich an Geschwindigkeit gewonnen, aber von der Schnelligkeit imperativer Strukturen sind wir noch ein gutes Stück entfernt.

Um weitere Effizienzgewinne erreichen zu können, müssen wir zumindest etwas in den Compiler und seine Implementierungsmechanismen hineinsehen. Allerdings hat sich die Situation in den letzten Jahren deutlich verbessert. Die Techniken, die wir im Folgenden beschreiben werden, mussten noch Anfang der 90er Jahre des vorigen Jahrhunderts in einer Doktorarbeit [128] sehr aufwendig bewiesen werden, indem auf subtile Weise über funktionale Programme und die Hoare-Logik ihrer imperativen Implementierungen simultan argumentiert wurde. Heute können wir diesen Bruch der Sprachebenen weitgehend vermeiden, indem wir z. B. das Mittel der *Monaden* (vgl. Kapitel 11) verwenden.

Zur Motivation der folgenden Diskussionen wollen wir noch einmal kurz das Problem analysieren. Die hocheffiziente imperative Implementierung von Datenstrukturen funktioniert normalerweise nur deshalb, weil man die Datenstrukturen selektiv ändert – also *unsicher* agiert. Das heißt, man erkauft sich Geschwindigkeit der Programmausführung mit fehleranfälligen und deshalb oft langwierigen und teuren Entwicklungsprozessen. Wir betrachten zur Illustration noch einmal *Queues*. Imperativ wird diese Datenstruktur oft so implementiert wie in Abbildung 13.1.

Es gibt je einen Pointer auf den Anfang und das Ende der Liste. Hinzufügen erfolgt am rechten Ende, Lesen und Wegnehmen am linken Ende. Damit

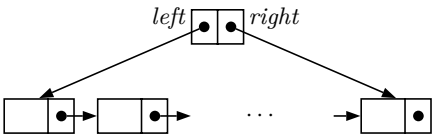


Abb. 13.1: Imperative Implementierung von Queues

haben alle interessanten Operationen den Aufwand $\mathcal{O}(1)$, aber in einem Programmfragment wie

$$\ldots f(queue :\cdot\ a,\ queue :\cdot\ b)\ldots$$

ist eine Verwendung dieser Implementierungstechnik nicht möglich, weil wir am Ende eine Verzweigung bräuchten. (Das ist unabhängig davon, ob wir in einer funktionalen oder einer imperativen Sprache programmieren.)

Wir werden im Laufe dieses Kapitels sehen, dass geeignete Implementierungstechniken (wie sie z. B. im OPAL-Compiler realisiert sind) zusammen mit der Queue-Darstellung aus Abschnitt 12.2 tatsächlich in der Lage sind, solche Situationen ohne Kopieren der Liste zu lösen. Trotzdem – dort, wo sie anwendbar ist – ist die Implementierungstechnik aus Abbildung 13.1 effizienter (wenn auch nur um konstante Faktoren) als unsere rein funktionalen Techniken.

13.1 Die Bedeutung von *Single-Threadedness*

Wir verwenden eine Terminologie, die aus dem Englischen übertragen wurde (sich allerdings in der deutschen Informatik nicht eingebürgert hat).

Definition (*single-threaded, ephemeral, multi-threaded, persistent*)

Wir nennen eine Datenstruktur

- ***flüchtige Datenstruktur*** (engl.: *ephemeral*, ***single-threaded***), wenn sie nur einmal benötigt wird und deshalb geändert werden darf;
- ***beständige Datenstruktur*** (engl.: *persistent*, ***multi-threaded***), wenn sie noch mehrmals benötigt wird und daher unverletzt erhalten werden muss.

Festzustellen, welche Datenstrukturen flüchtig sind, ist somit eine wichtige Compileraufgabe; denn für sie können die effizienten destruktiven Implementierungen eingesetzt werden, ohne dass die Sicherheit korrumpiert wird. Grundsätzlich gibt es für dieses Problem drei Vorgehensweisen:

- *Compilezeitanalysen*: Im Compiler sind Analyseroutinen vorhanden, die Single-Threadedness erkennen.
- *Sprachrestriktionen*: Die Programmiersprache enthält Notationen, mit denen der Programmierer Single-Threadedness ausdrücken kann.

- *Laufzeitanalysen*: Das System rechnet zur Laufzeit genug Information mit, um Single-Threadedness erkennen zu können.

Wir werden den ersten Punkt nur kurz ansprechen, weil er eigentlich Thema für ein Compilerbau-Buch wäre. Die beiden weiteren Punkte sind dagegen interessanter, weil sie auch dann, wenn man bei der Programmierung keinen direkten Einfluss nehmen kann, zum besseren Verständnis beitragen. Das ist insbesondere deshalb wichtig, weil man so eine bessere Einsicht in die Funktionsweise der Techniken des vorigen Kapitels erhält.

13.1.1 Die Analyse von *Single-Threadedness*

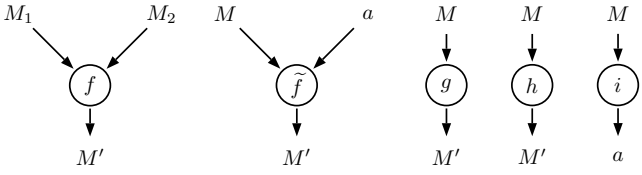
Wir betrachten als Beispiel eine Datenstruktur *Matrix* (also eine Struktur bei der Kopieren tunlichst vermieden werden sollte). Außerdem sei eine Operation $upd(M, i, x)$ gegeben, die die Matrix M am Index i (ein Paar, ein Tripel etc., je nach Dimension der Matrix) auf den Wert x ändert, genauer: eine neue Matrix erzeugt, die mit M überall außer an der Stelle i übereinstimmt. Die Operation $sel(M, i)$ erlaubt den Zugriff auf einzelne Matrixelemente.

TYPE *Matrix* α
FUN *upd*: *Matrix* $\alpha \times$ *Index* $\times \alpha \to$ *Matrix* α
FUN *sel*: *Matrix* $\alpha \times$ *Index* $\to \alpha$

Um die Probleme zu illustrieren, vergleichen wir zwei charakteristische Situationen. Seien dazu Funktionen der folgenden Art gegeben:

FUN f: *Matrix* $\alpha \times$ *Matrix* $\alpha \to$ *Matrix* α
FUN $\widetilde{f}$: *Matrix* $\alpha \times \alpha \to$ *Matrix* α
FUN g: *Matrix* $\alpha \to$ *Matrix* α
FUN h: *Matrix* $\alpha \to$ *Matrix* α
FUN i: *Matrix* $\alpha \to \alpha$

Der Datenfluss bei diesen Operationen ist in folgenden Graphen illustriert:



Betrachten wir jetzt drei Funktionen foo_1, foo_2 und foo_3, deren Datenfluss in Abbildung 13.2 illustriert ist. Dabei sind diejenigen Kanten, die *neue* Werte vom Typ *Matrix* α repräsentieren, schwarz gezeichnet, die anderen grau.

FUN foo_1: *Matrix* $\alpha \to$ *Matrix* α
DEF $foo_1(M) = f(g(M), h(M))$ $\quad --$ *M ist persistent*

FUN foo_2: *Matrix* $\alpha \to$ *Matrix* α
DEF $foo_2(M) = \widetilde{f}(g(M), i(h(M)))$ $\quad --$ *M ist persistent*

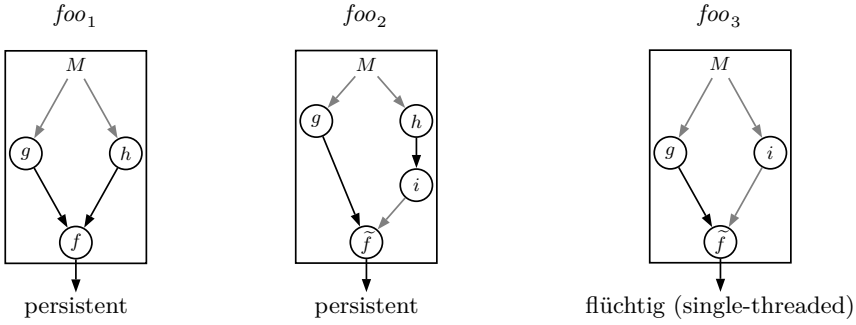


Abb. 13.2: Datenflussgraphen

FUN foo_3: $Matrix\ \alpha \to Matrix\ \alpha$
DEF $foo_3(M) = \widetilde{f}(g(M), i(M))$ $\quad -- \ M$ *ist flüchtig (single-threaded)*

Die Datenflussgraphen aus Abbildung 13.2 zeigen unmittelbar folgende Eigenschaften:

- Bei foo_1 ist die Matrix *persistent*. Das lässt sich auch textuell deutlicher machen, indem man LET-Klauseln benutzt:

$$\begin{array}{lll} \text{DEF } foo_1(M) = \text{LET} & M_1 = g(M) & -- \textit{ neue Matrix kreiert} \\ & M_2 = h(M) & -- \textit{ alte Matrix noch einmal benutzt} \\ \text{IN} & f(M_1, M_2) & \end{array}$$

- Auch foo_2 ist *persistent*, wie ebenfalls in einer sequenzialisierten Textversion zu sehen ist:

$$\begin{array}{lll} \text{DEF } foo_2(M) = \text{LET} & M_1 = g(M) & -- \textit{ neue Matrix kreiert} \\ & M_2 = h(M) & -- \textit{ alte Matrix noch einmal benutzt} \\ & a = i(M_2) & \\ \text{IN} & \widetilde{f}(M_1, a) & \end{array}$$

- Bei foo_3 ist die Matrix dagegen *flüchtig*:

$$\begin{array}{lll} \text{DEF } foo_3(M) = \text{LET} & a = i(M) & \\ & M_1 = g(M) & -- \textit{ neue Matrix kreiert} \\ \text{IN} & \widetilde{f}(M_1, a) & \end{array}$$

  Wenn M_1 eingeführt wird, wird M nicht mehr benötigt; also kann (im Compiler) der Speicherplatz von M für M_1 wiederverwendet werden – und das erlaubt natürlich auch das effizientere selektive Ändern.

Diese Beispiele illustrieren das generelle Prinzip: *Im Datenflussgraphen dürfen keine zwei schwarzen Kanten parallel zueinander liegen.* Denn in diesen Fällen ist es unmöglich, eine Sequenzialisierung zu finden, bei der M nicht noch benötigt wird.

Man beachte aber, dass auch in Situationen, die an sich single-threaded wären, durch ungeschickte Sequenzialisierung die Eigenschaft zerstört werden kann. Zum Beispiel darf unser obiges Beispiel foo_3 *nicht* so sequenzialisiert werden:

$$\begin{array}{lll} \text{DEF } foo_3(M) = \text{LET } M_1 = g(M) & -- \textit{ neue Matrix kreiert} \\ \qquad\qquad\qquad\quad a = i(M) & -- \textit{ alte Matrix noch einmal benutzt} \\ \qquad\quad \text{IN } \widetilde{f}(M_1, a) \end{array}$$

Jetzt wird M nach der Einführung von M_1 noch gebraucht.

Dieses Beispiel deutet das *zentrale Problem der Single-threaded-Analyse* an: Man muss unter allen möglichen Sequenzialisierungen eines Programms diejenige finden, die die Single-threaded-Eigenschaft besitzt (sofern eine solche existiert). Man kann die Frage noch etwas verallgemeinern: Man finde eine Sequenzialisierung, bei der die Single-threaded-Eigenschaft maximal häufig erfüllt ist (bei der also am seltensten kopiert werden muss).

Der Aspekt „unter allen möglichen Sequenzialisierungen“ zeigt das Dilemma: *Die Analyse ist im Allgemeinen exponentiell!* Deshalb werden wir diese Analysetechniken hier nicht weiter verfolgen, sondern uns auf die zugehörigen Sprach- und Implementierungstechniken konzentrieren.

13.1.2 Monaden garantieren Single-Threadedness

Wir hatten in Kapitel 11 das Konzept der *Monaden* eingeführt, allerdings eher als theoretisches Phänomen, mit dem sich die Nützlichkeit von Typklassen illustrieren ließ. Dabei sind Monaden ursprünglich aus ganz pragmatischen Gründen in die Funktionale Programmierung eingeführt worden: Sie sollten die Brücke zwischen funktionaler und imperativer Welt schlagen. (Wir werden darauf in Kapitel 17 noch einmal ausgiebig zurückkommen.)

Die Nützlichkeit der Monaden liegt darin, dass man mit ihren Operationen gar nicht anders als single-threaded programmieren *kann*. Diese Eigenschaft wird z. B. bei Ein-/Ausgabe essenziell gebraucht, um die Kommunikation mit der realen Welt adäquat in den Griff zu bekommen. Aber auch bei den Effizienzproblemen, die wir hier betrachten, ist Single-Threadedness ein entscheidender Aspekt.

Wir benutzen die *Zustands-Monade* aus Abschnitt 11.2.3 und zeigen, wie sich unsere obigen Beispielprogramme mit ihrer Hilfe monadisch umformulieren lassen. Die entsprechende Struktur ist in Programm 13.1 gezeigt. Da wir die Matrix als single-threaded garantieren wollen, müssen wir sie zum (versteckten) Zustand der Maschine machen. Mit Hilfe einer Funktion *init* können wir den Anfangszustand setzen. Die Funktion *upd* hat nur noch zwei Argumente, den Index und den Wert, und verändert den verborgenen Matrixzustand; sie hat kein sichtbares Resultat. Die Operation *sel* hat nur noch den Index als Argument und greift auf den verborgenen Matrixzustand zu; sie hat folglich einen sichtbaren Wert vom Typ α.

Programm 13.1 Die monadische „Matrix-Maschine“

```
STRUCTURE MatrixMonad[α: Type] = {                    -- "Matrix Machine"
  EXTEND Machine(Matrix α) RENAMING Com AS MaMa
  FUN init: Matrix → MaMa Void
  FUN upd: Index × α → MaMa Void
  FUN sel: Index → MaMa α
}
```

Auf der Basis dieser Struktur können wir jetzt Matrix-basierte Programme in monadische Form bringen. Dabei gelten folgende Übersetzungsschemata:

$f\colon Matrix \to Matrix$	wird zu	$f\colon MaMa\ Void$
$f\colon \beta \to Matrix \to Matrix$	wird zu	$f\colon \beta \to MaMa\ Void$
$f\colon \beta \to Matrix \to \gamma$	wird zu	$f\colon \beta \to MaMa\ \gamma$

Auf diese phänotypischen Situationen lassen sich auch alle anderen Funktionalitäten zurückführen, indem man gegebenenfalls Tupel umordnet und Currying anwendet. Wenn wir diese Schemata auf unsere obigen Funktionen anwenden, entstehen folgende neue Funktionen:

```
FUN f~: α → MaMa Void
FUN g: MaMa Void
FUN h: MaMa Void
FUN i: MaMa α
```

Man beachte, dass die Funktion $f\colon Matrix \times Matrix \to Matrix$ *nicht* transformiert werden kann – zumindest nicht, wenn Parameter vom Typ *Matrix* generell versteckt werden sollen.

Auf dieser Basis lässt sich dann von den drei obigen Funktionen nur foo_3 in monadische Form übertragen. Aus der rein funktionalen Form

```
FUN foo3: Matrix → Matrix        -- Originalversion
DEF foo3(M) = f~(g(M), i(M))
```

wird die monadische Form

```
FUN foo3: MaMa Void              -- monadische Version
DEF foo3 = i & λa • g & f~(a)
```

Da bei foo_1 und foo_2 jeweils persistente Matrizen vorkommen, ist eine monadische Version ausgeschlossen. Außerdem kommt noch hinzu, dass zumindest foo_1 die Funktion f benutzt, die monadisch nicht verfügbar ist.

Mit dieser Skizze des prinzipiellen Vorgehens wollen wir uns hier begnügen. Insbesondere lassen wir offen, ob diese Übersetzung vom Programmierer gemacht werden muss oder ob wir sie dem Compiler zutrauen können. (Zur Erinnerung: Die *Analyse* darüber, wo die Übersetzung erfolgen sollte, ist *ex-*

ponentiell.) Wir kommen allerdings im weiteren Verlauf des Kapitels noch einmal auf die Monadentechnik zurück.

13.1.3 Lineare Typen garantieren *Single-Threadedness*

Die im letzten Abschnitt diskutierten Probleme, insbesondere der exponentielle Analyseaufwand, legen nahe, einen Teil des Problems auf den Programmierer abzuwälzen. Denn dieser besitzt Wissen, das dem Compiler grundsätzlich fehlt: Oft folgt nämlich aus der Applikation, dass gewisse Strukturen unbedingt single-threaded sein müssen, während sich der Analyseaufwand bei anderen nicht lohnt, weil das Kopieren – selbst wenn es unnötig sein sollte – fast nichts kostet.

Allerdings muss man dem Programmierer die Chance geben, dieses Wissen auf bequeme Art dem Compiler mitzuteilen. Eine Möglichkeit dazu sind ***lineare Typen***. (In der Sprache CLEAN sind solche Typen als so genannte *Uniqueness types* enthalten [119].)

Nehmen wir als Beispiel eine Funktion *foo*: $T \times T \to T$. Wenn der Programmierer die Typisierung dieser Funktion folgendermaßen notiert

FUN *foo*: $T \times$ LINEAR $T \to T$

dann verlangt er, dass *foo* im zweiten Argument single-threaded ist. (Die Angabe für das Resultat kann man sich hier ersparen, weil es nur ein Resultat gibt.) Natürlich kann man auch einen ganzen Typ grundsätzlich linear machen:

TYPE $T_1 =$ LINEAR T

Das bedeutet, dass der Typ an allen Anwendungsstellen single-threaded sein muss, weshalb wir jetzt kurz schreiben können.

FUN *foo*: $T \times T_1 \to T_1$

Wenn mehrere gleich typisierte Argumente und Resultate jeweils gekennzeichnet werden sollen, brauchen wir sogar benannte lineare Typen:

TYPE $T_1 =$ LINEAR T
TYPE $T_2 =$ LINEAR T
FUN *bar*: $T_1 \times T_2 \times T \to T_2 \times T_1$

In der Literatur gibt es Kalküle für solche linearen Typen, die vor allem auf Arbeiten von Girard über *lineare Logik* zurückgehen (z. B. [57, 58, 59]); weitere Informationen findet man auch in [118]. Die Details gehen aber über den Rahmen unseres Buches hinaus. Wir wollen hier nur einen groben Eindruck von der prinzipiellen Vorgehensweise vermitteln.

Typkalküle werden meistens in der Form so genannter *Typisierungsregeln* beschrieben. Klassische Beispiele für diese Notation sind die folgenden beiden Regeln, die die Typisierung des IF-Ausrucks und der Funktionsapplikation definieren:

$$\textit{IF}\quad \frac{\mathcal{A} \vdash e_1 : \mathit{Bool}, \quad \mathcal{A} \vdash e_2 : T, \quad \mathcal{A} \vdash e_3 : T}{\mathcal{A} \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \text{ FI} : T}$$

$$\textit{Apply}\quad \frac{\mathcal{A} \vdash f : (T_1 \to T_2) \quad \mathcal{A} \vdash e : T_1}{\mathcal{A} \vdash f(e) : T_2}$$

Diese Notation ist folgendermaßen zu lesen: Wenn in einem Kontext $\mathcal{A}$ (d. h. in einer gegebenen Typisierung der vorkommenden Namen) e_1 den Typ *Bool* hat und e_2 und e_3 jeweils den Typ T haben, dann hat der gesamte IF-Ausdruck den Typ T. Analog heißt die zweite Regel: Wenn im Kontext $\mathcal{A}$ die Funktion f den Funktionstyp $(T_1 \to T_2)$ hat und wenn der Argumentausdruck e den Typ T_1 hat, dann hat die Funktionsapplikation $f(e)$ insgesamt den Resultattyp T_2.

In dieser Form werden alle Konstrukte einer Sprache beschrieben. In *klassischen Sprachen*, also Sprachen ohne lineare Typen, hat man insbesondere die folgenden beiden Typregeln:

$$\textit{Weakening}\ \frac{\mathcal{A} \vdash e : T}{\mathcal{A}, x : T_1 \vdash e : T} \qquad \textit{Contraction}\ \frac{\mathcal{A}, x : T_1, y : T_1 \vdash e : T}{\mathcal{A}, z : T_1 \vdash e[z/x, z/y] : T}$$

Die *Weakening*-Regel erlaubt es, Werte zu ignorieren (und ermöglicht damit *Lazy evaluation*), und die *Contraction*-Regel erlaubt es, Werte zu duplizieren.

Genau diese beiden Aktivitäten sind es aber, die der Linearität entgegenlaufen. Deshalb muss man neue Arten von Typen einführen: ***Lineare Typen*** – geschrieben als $\widehat{T}$ – erlauben *kein* Weakening oder Contraction, während beides für nichtlineare Typen erlaubt bleibt. Dafür kommen zwei neue Regeln hinzu: *Promotion* und *Dereliction*.

$$\textit{Weakening}\ \frac{\widehat{\mathcal{A}} \vdash e : \widehat{T}}{\widehat{\mathcal{A}}, x : T_1 \vdash e : \widehat{T}} \qquad \textit{Contraction}\ \frac{\widehat{\mathcal{A}}, x : T_1, y : T_1 \vdash e : \widehat{T}}{\widehat{\mathcal{A}}, z : T_1 \vdash e[z/x, z/y] : \widehat{T}}$$

$$\textit{Promotion}\ \frac{\mathcal{A} \vdash e : \widehat{T}}{\mathcal{A} \vdash e : T} \qquad \textit{Dereliction}\ \frac{\widehat{\mathcal{A}}, x : \widehat{T_1} \vdash e : \widehat{T}}{\widehat{\mathcal{A}}, x : T_1 \vdash e : \widehat{T}}$$

Die *Promotion*-Regel besagt, dass ein linearer Typ immer auch als nichtlinear betrachtet werden kann. Bedingung dafür ist aber, dass keine seiner freien Variablen linear ist. (In diesem Fall muss auch e linear bleiben.)

Die *Dereliction*-Regel besagt, dass eine Variable, die genau einmal in e vorkommt, auch nichtlinear sein darf. Entsprechendes sagt die *Weakening*-Regel für Variablen, die überhaupt nicht in e vorkommen, und die *Contraction*-Regel für Variablen, die mehrfach in e vorkommen.

Wir können hier nicht weiter auf die Subtilitäten dieser Art von Typkalkülen eingehen und verweisen deshalb wieder auf die schon erwähnte Literatur.

Wir gehen davon aus, dass – analog zur Situation bei Laziness – dem Programmierer mit dem Schlüsselwort LINEAR die Möglichkeit eröffnet wird, Single-Threadedness als Forderung auszudrücken. Es ist dann die Aufgabe des Compilers – gemäß den oben skizzierten Typisierungsregeln – nachzuprüfen, ob die vom Programmierer behauptete Linearität in der Tat zutrifft. Das ist wesentlich einfacher und effizienter zu realisieren als die Analyse, *wo* Linearität vorliegt.

13.2 *A Dag For All Heaps (Reference-Counting)*

Wenn zur Compilezeit – mit oder ohne Hilfe des Programmierers – keine befriedigende Lösung des Single-Threadedness-Problems gefunden werden kann, dann bleiben nur noch *Laufzeitanalysen*. Natürlich bedeutet das a priori einen gewissen Effizienzverlust, da diese Analysen kontinuierlich mitgerechnet werden müssen. Aber der Zusatzaufwand ist im Allgemeinen um Größenordnungen geringer, als es unnötige Kopieraktionen wären. (Und er ist eben der Preis dafür, dass wir garantierte Sicherheit haben.)

13.2.1 Ein Dag-Modell für das Datenmanagement

Bekanntlich werden alle über rekursive Typen definierten Strukturen (also Listen, Bäume etc.) im Speicher letztlich als verzeigerte Konglomerate von Zellen repräsentiert. Daher ist eine Modellierung über Graphen adäquat:

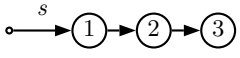
- Die ***Knoten*** repräsentieren die Zellen.
- Die ***Kanten*** repräsentieren die Pointer zwischen den Zellen.

Zu jedem Zeitpunkt während der Laufzeit eines Programms bilden alle seine Datenstrukturen gemeinsam einen großen Graphen, genauer: einen *gerichteten azyklischen Graphen* (engl.: *directed acyclic graph*, kurz *Dag*). Denn Strukturen, die über Konstruktoren von (rekursiven) Typdeklarationen aufgebaut sind, können keine Zyklen enthalten. (Wenn man allerdings unendliche Strukturen zulässt, ändert sich das; dann braucht man aufwendigere Techniken.) Wir betrachten als einfaches Beispiel den Listentyp:

$$\begin{array}{lcl} \text{TYPE } \mathit{Seq}\ \alpha & = & \_ \cdot\!: \_\ (\mathit{ft} = \alpha,\ \mathit{rt} = \mathit{Seq}\ \alpha) \\ & \mid & \mathit{Empty} \end{array}$$

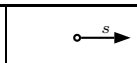
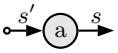
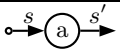
Elemente dieses Types stellen wir graphisch folgendermaßen dar:

$$\text{LET } s = 1 \cdot\!: 2 \cdot\!: 3 \cdot\!: \Diamond \text{ IN } \ldots$$



In dieser graphischen Modellierung entspricht jedes Element vom Typ *Seq* einer Kante; wir benutzen daher die Terminologie:

Der Wert s ist eine Kante (vom Typ Seq).

Operation	vorher	nachher	Erläuterung
$s' = a \cdot\!: s$			Es wird ein neuer Knoten kreiert, der mit a belegt wird und an den die Kante s angehängt wird. Das Ergebnis der Operation ist eine Kante s' auf den neuen Knoten.
$a = ft(s)$		a	Da die Operation *ft* fertig ist, wird das Argument (d. h. diese Kante) nicht mehr gebraucht. Resultat ist der Inhalt des Knotens, auf den die Kante s zeigt.
$s' = rt(s)$			Da die Operation *rt* fertig ist, wird das Argument s nicht mehr gebraucht. Ergebnis ist eine Kopie der Ausgangskante s' des Knotens.

Tab. 13.1: Regeln für das Speichermodell

In diesem Modell haben die Konstruktoren und Selektoren die üblichen Realisierungen, die wir in Tabelle 13.1 skizzieren.

Einige Erläuterungen zu diesem Modell:

- Die Kanten modellieren die Zeiger der späteren Maschinenebene. Sie sind eigenständige Elemente im Graphen und besitzen insbesondere einen Start- und einen Endknoten.
- In Programmtermen wie $ft(s)$, $rt(s)$ oder $a \cdot\!: s$ modelliert die Kante s den Zeiger aus dem Execution stack in den Heapspeicher. Wenn Operationen fertig verarbeitet sind, werden ihre Daten aus dem Stack entfernt. Deshalb muss in unserem Modell am Ende jeder Operation die Argumentkante aus dem Dag entfernt werden.

Wenn wir diese Überlegungen in ein konkretes Programmiermodell fassen wollen, müssen wir Folgendes tun:

1. Zunächst brauchen wir eine Struktur mit einem Typ *Dag* für gerichtete azyklische Graphen. Ferner müssen Operationen für das Hinzufügen, Löschen und Ändern von Knoten und Kanten verfügbar sein.
2. Dann führen wir eine Zustands-Monade ein, bei der der *Dag* zum verborgenen Zustand wird. Wir nennen diese Monade *Heap*.
3. Schließlich übersetzen wir die *Seq*-Operationen (genauer: ihre monadischen Versionen) in entsprechende Heap-Operationen gemäß den Regeln in Tabelle 13.1.

(1) Gerichtete azyklische Graphen

Wir verwenden hier keine allgemeine Graph-Struktur, sondern eine sehr spezielle Variante, die genau auf unsere Implementierungsbedürfnisse zugeschnitten ist. Daher sehen einige Operationen etwas ungewöhnlich aus (was gleich noch näher begründet wird). Wir beschränken uns auf die Angabe der Signatur.

Programm 13.2 Der Typ *Dag*

```
SIGNATURE DirectedAcyclicGraph = {
  TYPE Dag                                              -- Graphen
  TYPE Arc α                                            -- Kanten
  FUN attach: Arc α × Arc β → Dag → (Dag × Arc α)       -- Kante anfügen
  FUN copy: Arc α → Dag → (Dag × Arc α)                 -- Kopie der Kante hinzufügen
  FUN delete: Arc α → Dag → Dag                         -- Kante löschen
  FUN content: Arc α → Dag → α                          -- Inhalt des Knotens
  FUN new: α → Dag → (Dag × Arc α)                      -- neuen Knoten hinzufügen
  FUN next: Arc α → Dag → Arc β                         -- nächste Kante
}
```

Was hier etwas ungewöhnlich erscheint, ist das Fehlen eines Typs *Node*. Der Grund liegt darin, dass wir wirklich das Verhalten eines Heap-Speichers modellieren wollen, wie man ihn im Compilerbau verwendet. Dabei wird ein Knoten grundsätzlich nur über Referenzen (also Pointer) angesprochen – und die entsprechen in unserem Modell den Kanten. Mit anderen Worten: wenn wir einen Knoten brauchen, geben wir eine Kante an, die auf ihn führt.

- Bei $G.attach(a, b)$ wird im Dag G die Kante b an den Zielknoten von a angefügt. (Die Kante b hat im Allgemeinen also einen neuen Startknoten, behält aber ihren Zielknoten.) Das Ergebnis ist die Kante a.
- Bei $G.copy(a)$ wird dem Dag eine Kopie der Kante a hinzugefügt (also eine weitere Verbindung vom Start- zum Zielknoten).
- Bei $G.delete(a)$ wird die Kante a aus dem Dag gelöscht. (Damit kann unter Umständen der Zielknoten unerreichbar werden.)
- Mit $G.content(a)$ wird der Inhalt des Zielknotens von a geliefert.
- Mit $G.new(x)$ wird ein neuer Knoten mit Inhalt x erzeugt und eine neue Kante a, die auf diesen Knoten zeigt. Als Ergebnis wird diese Kante geliefert.
- Mit $G.next(a)$ wird diejenige Kante geliefert, die vom Zielknoten von a ausgeht. (Diese Operation ist ganz auf unser Beispiel der Sequenzimplementierung abgestimmt, bei der es in jedem Knoten nur eine Ausgangskante gibt. Für andere Strukturen wie Bäume etc. bräuchten wir hier etwas aufwendigere Operationen.)

Man beachte, dass wir bei diesem Modell die Kanten als eine eigenständige Menge betrachten. Sie haben im Allgemeinen einen Start- und einen Zielknoten, aber das muss nicht sein. So wird z. B. bei *new* eine Kante ohne Startknoten generiert[1] und bei *attach* der Startknoten einer Kante geändert.

(2) Die Heap-Monade

Der oben eingeführte Typ *Dag* wird als Basis einer geeigneten Zustands-Monade genommen. In Abschnitt 11.2.3 haben wir Zustands-Monaden eingeführt, deren wesentliches Merkmal ein interner Zustand, der so genannte *Hidden state*, ist. Dieser interne Zustand ist jetzt vom Typ *Dag*. Programm 13.3 enthält die entsprechende Struktur. Zur Erinnerung: Der Monadentyp $M\ \alpha$ selbst wird bei der Zustands-Monade in $Com\ \alpha$ umbenannt (vgl. Programm 11.6 in Abschnitt 11.2.3).

Programm 13.3 Die Heap-Monade

```
STRUCTURE Heap = {                                  -- Der "Speicher"
  EXTEND Machine(Dag)
  FUN attach: Arc α × Arc β → Com(Arc α)            -- Kante hinzufügen
  FUN copy: Arc α → Com(Arc α)                      -- Kante kopieren
  FUN delete: Arc α → Com Void                      -- Kante löschen
  FUN content: Arc α → Com α                        -- Inhalt des ref. Knotens
  FUN new: α → Com(Arc α)                           -- neuen Knoten hinzufügen
  FUN next: Arc α → Com(Arc β)                      -- nächste Kante
}
```

Die spezifischen Operationen dieser Struktur sind Umschreibungen der *Dag*-Operationen. Das geht ganz systematisch:

- Aus $(\alpha \to Dag \to \beta)$ wird $(\alpha \to Com\ \beta)$.
- Aus $(\alpha \to Dag \to (Dag \times \beta))$ wird $(\alpha \to Com\ \beta)$.
- Aus $(\alpha \to Dag \to Dag)$ wird $(\alpha \to Com\ Void)$.

(3) Implementierung der Seq-Operationen

Jetzt werden auch die elementaren *Seq*-Operationen in monadische Form gebracht. Dies geschieht nach demselben Prinzip wie bei der Umwandlung der *Dag*-Operationen. Programm 13.4 enthält die entsprechenden Operationen. Zur besseren Lesbarkeit benennen wir den Typ *Arc* in *Seq* um. Die Definitio-

[1] Da diese Kante im Programm verwendet wird, modelliert sie genau die Tatsache, dass es jetzt einen Zeiger vom Programm (dem so genannten *Execution stack*) in den Heap gibt. Wenn z. B. im Programm steht LET $a = new(x)$, dann entspricht der Wert a diesem Zeiger. Solche Zeiger müssen beim Reference counting genauso mitgezählt werden wie die Heap-internen Zeiger.

Programm 13.4 Monadische Form der *Seq*-Operationen

```
IMPORT Heap RENAMING Arc AS Seq
FUN _ ·: _: α × Seq → Com Seq
FUN ft: Seq → Com α
FUN rt: Seq → Com Seq

DEF (x ·: s) = new(x) → s_new & attach(s_new, s)
DEF ft s = content(s) → a & delete(s) & yield(a)
DEF rt s = next(s) → s_next & delete(s) & yield(s_next)
```

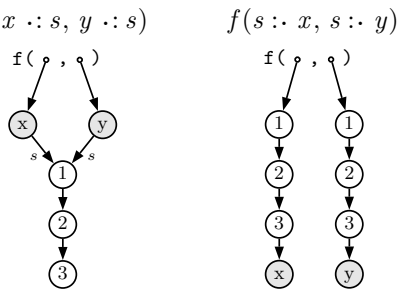
nen dieser Operationen sind eine direkte Umsetzung der Regeln aus Tabelle 13.1 mit Hilfe der gegebenen *Heap*-Operatoren.

- Bei $x \cdot: s$ wird zunächst ein neuer Knoten mit Inhalt x erzeugt und eine Kante s_{new}, die auf ihn zeigt. Dann wird die Kante s an diesen Knoten angehängt. (Sie erhält ihn also als neuen Startknoten.) Man beachte, dass aufgrund der Definition von *attach* die neue Kante s_{new} als Ergebnis abgeliefert wird; deshalb ist $yield(s_{new})$ hier nicht nötig.
- Bei $(ft\ s)$ wird der Inhalt des entsprechenden Knotens beschafft und als Ergebnis geliefert. Die Kante s wird gelöscht, weil sie einen Zugriff aus dem Programm in den Heap repräsentiert, der nach dem Ende der Operation *ft* nicht mehr existiert.
- Bei $(rt\ s)$ wird die Kante auf den Nachfolgerknoten als Ergebnis geliefert. Die Kante s selbst wird auch hier gelöscht.

Aus Gründen der Lesbarkeit werden wir im Folgenden nicht nur auf dieser – doch sehr technisch überfrachteten – programmiersprachlichen Ebene arbeiten, sondern uns darauf konzentrieren, die essenziellen Ideen graphisch zu erläutern.

Stack versus Queue. Auf der Basis dieser Dag-Modellierung betrachten wir jetzt noch einmal den Unterschied zwischen den *push*-Operationen bei Stacks und Queues. In den beiden Beispielprogrammen liegt *keine* Single-Threadedness vor. Die Diagramme illustrieren jeweils die Situation nach Auswertung der beiden Argumentausdrücke.

Im Fall (a) ist kein Kopieren der Liste s nötig; aber die Tatsache, dass der Knoten (1) den Indegree 2 hat, zeigt, dass die Liste s „doppelt zählt“.



(a) Stack (b) Queue

Im Fall (b) dagegen ist eine Duplizierung der Liste nötig, da die Verzweigung erst am Ende stattfindet.

Dass die in der obigen Tabelle 13.1 skizzierten Regeln das Bild (a) ergeben, sieht man sofort ein. Aber wie entsteht das Resultat von Bild (b)? Damit beschäftigen sich die folgenden Abschnitte.

Man beachte übrigens eine wichtige Asymmetrie:

- Der *Outdegree* eines Knotens, also die Zahl (und die Typen) der Kanten, die aus ihm herausgehen, ist im Typ festgelegt.
- Der *Indegree*, also die Zahl der Kanten, die auf ihn zeigen, ist völlig offen.

Warum *„Reference-Counting“*?

Wir haben diese Methode unter dem Schlagwort „*Reference-Counting*“ eingeführt, obwohl in unserem Dag-Modell weder Referenzen noch Zähler explizit auftauchen. Aber offensichtlich sind die Kanten nur eine unbedeutende Abstraktion der klassischen Pointer, durch die sie im Compiler dann natürlich dargestellt werden.

Und die Zähler werden gebraucht, um das Problem der *Indegrees* zu lösen. Denn es ist im Modell leicht zu sagen, dass der Eingangsgrad eines Knotens 0 ist. Im Programm funktioniert das nur, wenn der Knoten kontinuierlich diese Zahl mitrechnet. Das heißt, wenn im Modell eine Kante verschwindet, muss im konkreten Code der Zähler reduziert werden, und wenn eine Kante erzeugt wird, muss der Zähler erhöht werden.

Diese Buchhaltung bewirkt natürlich einen gewissen Effizienzverlust. Deshalb versucht man ihn durch geeignete Analyse- und Optimierungstechniken so weit wie möglich zu reduzieren. Zum Beispiel kann man bei der bekannten *length*-Funktion auf Listen die Zähler einfach ignorieren, weil ja keine neue Liste erzeugt wird, sondern nur eine vorhandene traversiert wird.

Anmerkung: Es ist eine reizvolle Spekulation, ob man mit Hilfe von abhängigen Typen das Reference-Counting so in die Knotentypen integrieren könnte, dass eine kombinierte Compilezeit- und Laufzeitoptimierung ermöglicht wird. Aber das ist heute eine offene Forschungsfrage.

13.2.2 Sicheres Management persistenter Strukturen

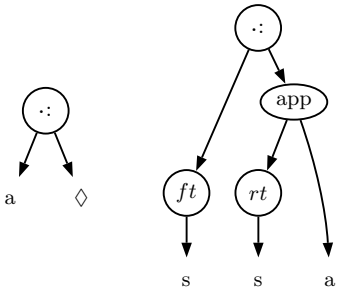
Wir demonstrieren die Strategie der Datenimplementierung im Dag-Modell anhand eines kleinen Beispiels. Dabei konzentrieren wir uns auf diejenige Operation, die am meisten Ärger macht, nämlich die *Append*-Operation von Queues, bei der das Anfügen am falschen Ende der Liste erfolgt. Aus Gründen der Lesbarkeit schreiben wir jetzt $app(s, a)$ anstelle von „$s :\cdot a$“.

Programm 13.5 zeigt die übliche funktionale Definition der Funktion *app* und die zugehörige monadische Version. Diese monadische Form entsteht im Wesentlichen dadurch, dass wir die monadischen Varianten von *ft*, *rt* und $\cdot:$ verwenden. Man beachte, dass bei $new(a)$ die neue Kante als Ergebnis abgeliefert wird.

Programm 13.5 Die Append-Operation *app* funktional und monadisch

Append in funktionaler Form:

FUN app: $Seq\ \alpha \times \alpha \to Seq\ \alpha$
DEF $app(\Diamond,\ a) = a \cdot\!: \Diamond$
DEF $app(s,\ a) = ft(s) \cdot\!: app(rt(s),\ a)$



Programmtext Programmgraphen

Append in monadischer Form:

FUN app: $Seq\ \alpha \times \alpha \to Com(Seq\ \alpha)$
DEF $app(\Diamond, a) = new(a)$
DEF $app(s, a) =$ $copy(s) \to s'$ &
$ft(s) \to x$ &
$rt(s') \to s_1$ &
$app(s_1, a) \to s_2$ &
$(x \cdot\!: s_2) \to s_3$ & $yield(s_3)$

Am Anfang der Funktion $app(s, a)$ kreieren wir eine Kopie des Kantenparameters, so dass jede Kante nur einmal im Rumpf verwendet wird. (Das modelliert letztlich das Reference counting.)

Auf der Basis dieser Definition studieren wir jetzt die Funktionsweise des Dag-Modells anhand eines kleinen Beispiels, das aber trotz seiner Einfachheit die entscheidenden Komplikationen zeigt.

Beispiel 13.1 (Funktionsweise des Dag-Speichermodells)

Wir betrachten ein kleines Programmfragment, in dem die *app*-Operation zweimal benutzt wird, so dass *keine* Single-Threadedness vorliegt.

$f(\ app(s, x),\ app(s, y)\ )$ WHERE $s = \langle 1, 2, 3\rangle$

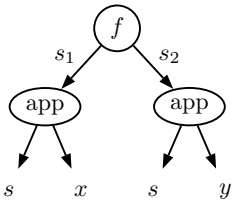
In monadischer Form sieht das folgendermaßen aus:

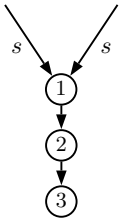
$copy(s) \to s'$ &
$app(s, x) \to s_1$ & $app(s', y) \to s_2$ & $f(s_1, s_2)$

(1) Wir haben folgende Ausgangssituation: Der Programmgraph hat zwei Kanten, die auf die Liste zeigen. Deshalb gibt es zu Beginn der Ausführung

unseres kleinen Programmfragments zwei s-Kanten im Dag. (Wir nehmen hier an, dass s sonst nirgends mehr vorkommt.)

$f(app(s,x), app(s,y))$



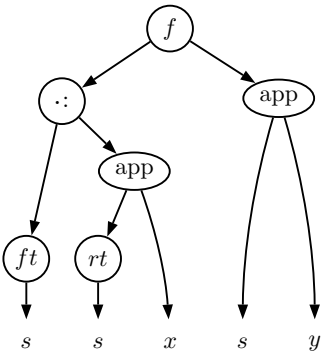


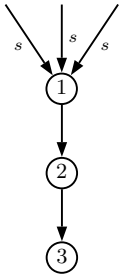
(1)

Programmgraph Dag-Speicher

(2) Jetzt muss eine der beiden *app*-Operationen ausgeführt werden. Semantisch lässt sich der Aufruf einer Funktion bekanntlich so modellieren, dass der Funktionsrumpf eingesetzt wird („Unfolding“). Wenn wir den Rumpf von *app* (genauer: den Rekursionszweig) anstelle des linken Aufrufs $app(s,x)$ einsetzen, ergibt sich:

$f(ft(s) \cdot\!: app(rt(s),x), app(s,y))$





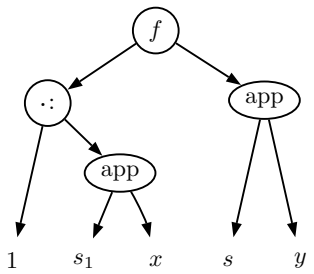
(2)

Programmgraph Dag-Speicher

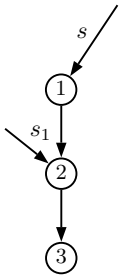
Die drei Verwendungen von s im Programmgraphen führen zu drei entsprechenden Kanten im Dag. Das lässt sich auch operational nachvollziehen: Sobald der Aufruf $app(s,x)$ ausgeführt wird, ist das Argument nicht mehr da und eine der beiden s-Kanten aus (1) verschwindet. Da aber im Programmgraph von *app* die Kante s zweimal gebraucht wird, entstehen zwei neue s-Kanten im Dag; damit gibt es dort jetzt insgesamt drei Kanten.

(3) Die Ausführung der primitiven Operationen *ft* und *rt* vereinfacht den Programmgraphen entsprechend: Nach der Ausführung von *ft* und *rt* sind zwei der drei s-Kanten wieder verschwunden. Dafür ist eine weitere s_1-Kante entstanden, mit der *app* jetzt rekursiv aufgerufen wird.

$f(1 \cdot\!: app(s_1, x), app(s, y))$



Programmgraph

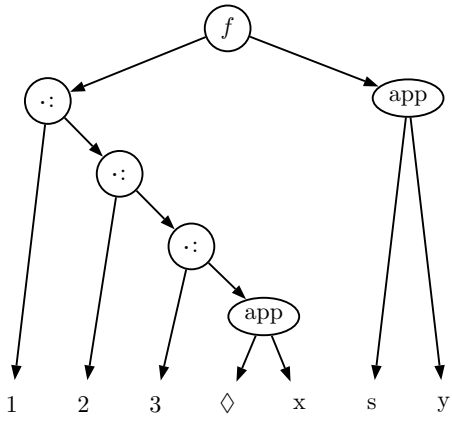


Dag-Speicher

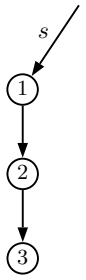
(3)

(4) Auf diese Weise arbeitet sich das Programm weiter in die Rekursion hinein, bis das Ende der Liste s erreicht ist.

$f(1 \cdot\!: 2 \cdot\!: 3 \cdot\!: app(\Diamond, x), app(s, y))$



Programmgraph



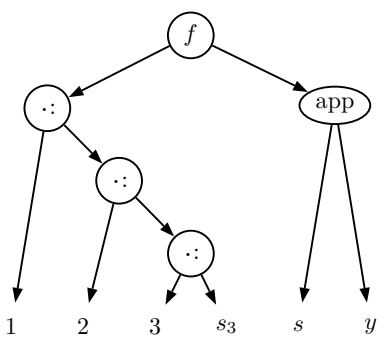
Dag-Speicher

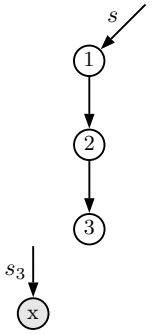
(4)

Man beachte, dass aus dem linken Zweig von f keine einzige Kante mehr in den Dag hineinführt. (Alle benötigten Daten stehen jetzt im Rekursionskeller, bzw. – aus semantischer Sicht – im Programmgraphen.)

(5) In dieser Situation wird der Terminierungszweig von *app* ausgeführt. Dieser besteht in der Ausführung des Konstruktors $x \cdot\!: \Diamond$. Das bedeutet nach unseren Regeln, dass ein neuer Knoten kreiert wird und als Resultat eine Kante auf diesen Knoten abgeliefert wird (die wir hier mit s_3 bezeichnet haben). Eine Kante auf die leere Liste führen wir nicht ein. (Das entspricht der Verwendung von *null*-Pointern in imperativen Sprachen.)

$f(1 \cdot\!: 2 \cdot\!: 3 \cdot\!: x \cdot\!: \Diamond, app(s, y))$

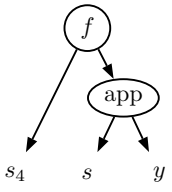


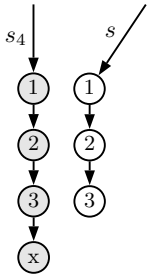


(5)

Programmgraph Dag-Speicher

(6) Jetzt müssen der Reihe nach die Konstruktoren „ $\cdot\!:$ “ ausgewertet werden, wodurch jeweils neue Knoten erzeugt und entsprechend miteinander verbunden werden. Am Ende ist das Ergebnis die Kante s_4 und wir haben folgende Konstellation:





(6)

Programmgraph Dag-Speicher

Damit ist die Verarbeitung des linken Aufrufs $app(s, x)$ abgeschlossen.

Das Beispiel zeigt, dass bei persistenten Strukturen die Implementierung sicher ist: Da die alte Struktur noch gebraucht wird, bleibt sie erhalten. Anders ausgedrückt: *Wenn Kopieren unvermeidlich ist, erfolgt es auch.*

13.2.3 Dynamische Erkennung von Single-Threadedness

Wenn im obigen Beispiel die Konfiguration (6) erreicht ist, erfolgt der rechte Aufruf $app(s, y)$. Jetzt liegt aber eine veränderte Situation vor:

Die Liste s ist jetzt single-threaded!

In einer naiven Implementierung würde trotzdem der gleiche Prozess wie für den vorigen Aufruf ablaufen. Dabei würde eine weitere Kopie der Liste erzeugt werden. Danach gäbe es keinen Zeiger auf die Originalliste mehr, die deshalb über kurz oder lang vom *Garbage Collector* eingesammelt würde.

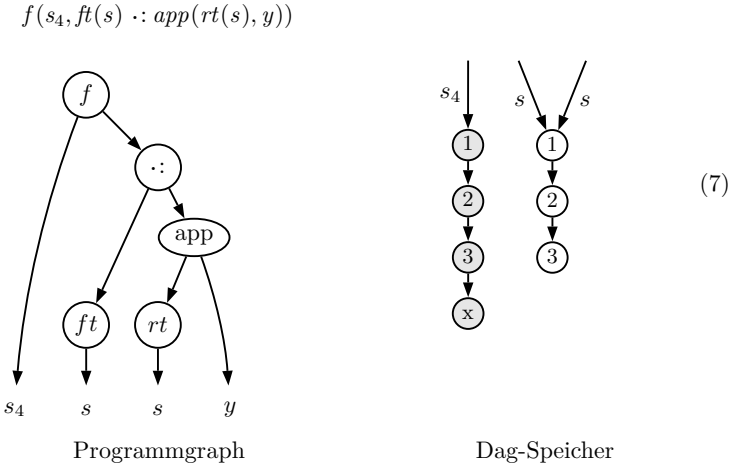
Eine solche naive Implementierung wird aber in modernen Compilern vermieden; stattdessen werden (z. B. im OPAL-Compiler) Techniken eingesetzt, die zur Laufzeit erkennen, ob eine Struktur single-threaded ist. Wenn ja, wird eine effizientere Methode für die Verwaltung des Speicherplatzes eingesetzt.

Das wollen wir jetzt studieren, indem wir unser obiges Beispiel fortsetzen. Dabei betrachten wir zuerst den naiven Ansatz, der die Umsetzung ganz schematisch macht und deshalb unnötig ineffizient wird. Auf diesen Beobachtungen aufbauend werden wir anschließend Optimierungsmöglichkeiten studieren.

Beispiel 13.2 (Laufzeitbehandlung von Single-Threadedness)

---

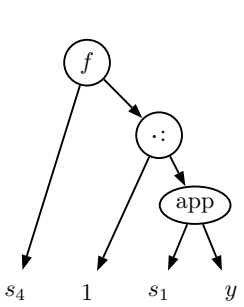
(7) In der Situation (6) wenden wir Unfolding auf den verbliebenen Aufruf $app(s, y)$ an. Wie auch schon früher entstehen so zwei s-Kanten in den Dag hinein.



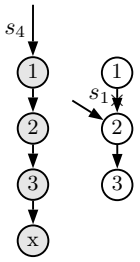
(8) Als Nächstes werden die primitiven Operationen *ft* und *rt* ausgeführt, wodurch die s-Kanten verschwinden. Dadurch hat aber der Knoten (1) den Indegree 0! Und das heißt, er ist Garbage.

Hier ist nun ein erster entscheidender Punkt für die Effizienzsteigerung: Wenn ein Knoten Garbage ist, dann sollten von ihm auch keine Kanten mehr ausgehen (weil er sonst unter Umständen andere Knoten daran hindert, auch als Garbage erkannt zu werden.)

$f(s_4, 1 \cdot\!: app(s_1, y))$



Programmgraph

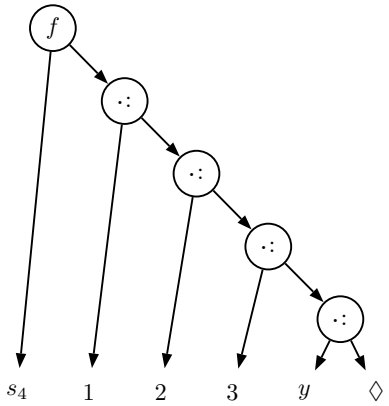


Dag-Speicher

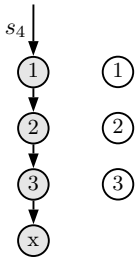
(8)

(9) Wie schon beim vorigen Beispiel erhalten wir auf diese Weise am Ende der Rekursion eine Situation, bei der keine einzige Kante mehr aus dem Programmgraphen in den Dag hinein zeigt. Im Unterschied zum letzten Beispiel gibt es aber auch keine anderen Pfade mehr, auf denen die Knoten erreichbar wären – sie sind allesamt Garbage.

$f(s_4, 1 \cdot\!: 2 \cdot\!: 3 \cdot\!: y \cdot\!: \Diamond)$



Programmgraph

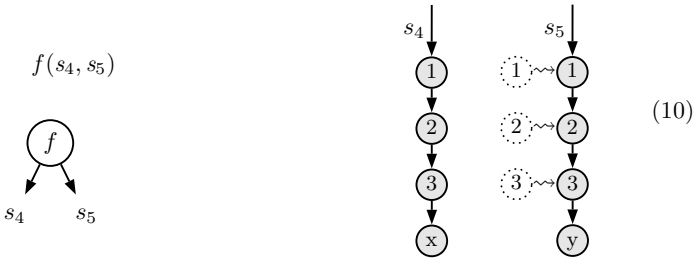


Dag-Speicher

(9)

(10) Jetzt werden die hängenden Inkarnationen der Rekursion abgearbeitet, d. h. die Konstruktoren „ $\cdot\!:$ “ ausgeführt. Im naiven Ansatz würden diese jeweils neue Zellen generieren; aber eine bessere Lösung besteht natürlich

darin, die gerade freigewordenen Zellen nicht wegzuwerfen, sondern für die „neuen“ Zellen gleich wieder zu recyceln.



Programmgraph Dag-Speicher

Damit ist das Beispiel abgeschlossen. (Die Funktion f betrachten wir nicht mehr.)

Im Folgenden wollen wir noch kurz betrachten, wie dieses Recycling funktioniert. Es gibt grundsätzlich zwei Möglichkeiten:

- Man kann eine relative schematische Lösung wählen, bei denen alle Knoten, sobald sie als Garbage erkannt sind, in eine *Freispeicherliste* wandern (*dealloc*). Von dieser Freispeicherliste werden bei Bedarf neu zu generierende Knoten beschafft (*alloc*).
  In unserem Beispiel führt diese Taktik dazu, dass die gerade freigegebenen Knoten – aufgrund des Stack-artigen Verhaltens der Freispeicherliste – sofort wieder herangezogen werden.
  Diese Situation kann man (sehr tief innen) im Compiler zu einer entscheidenden Verbesserung nutzen. Mittels so genannter *Peephole optimization* kann man versuchen, die Instruktionen so zu verschieben, dass jeweils Paare von *dealloc*- und *alloc*-Operationen direkt nebeneinander stehen. Diese heben sich dann gegenseitig auf und der Overhead des Umwegs über die Freispeicherliste wird eingespart.
- Eine zweite Lösung betrachtet das Problem auf einem höheren Level und ist somit nicht auf das Funktionieren der *Peephole optimization* angewiesen. Da sie methodisch interessanter ist, werden wir sie im Folgenden etwas genauer betrachten.

13.2.4 *„Tail-Rekursion modulo cons“*

Wie wir gesehen haben, arbeitet man sich bei einer Operation wie *app* bei der Ausführung zunächst in die Rekursion hinein, um dann beim „Zurückklappern“ alle hängenden Konstruktoroperationen „ $\cdot$: “ auszuführen. (Das ist das übliche Verhalten bei linear rekursiven Funktionen; vgl. Abschnitt 1.4.) Bei langen Listen heißt das, dass alle Listenelemente im Rekursionkeller zwischengespeichert sind, der damit sehr lang (und ineffizient) wird.

Deshalb sollten wir versuchen, mit den Techniken aus Abschnitt 1.4 die Funktion *app* in Tail-rekursive Form umzuwandeln. Leider ist aber der Operator „ $\cdot{:}$ “ nicht assoziativ, so dass wir uns etwas anderes ausdenken müssen.

Versuch 1: *Continuations*

Wir können den universellen Trick aus Abschnitt 1.4.2 verwenden und mit Continuations arbeiten. Wenn wir diese Technik auf die funktionale Form von *app* aus Programm 13.5 anwenden, entsteht folgendes System von Definitionen:

FUN app: $Seq\ \alpha \times \alpha \to Seq\ \alpha$
FUN $\widetilde{app}$: $Seq\ \alpha \times \alpha \times (Seq\ \alpha \to Seq\ \alpha) \to Seq\ \alpha$
DEF $app(s,\ a) = \widetilde{app}(s,\ a,\ id)$
DEF $\widetilde{app}(\lozenge,\ a,\ \Gamma) = \Gamma(\ a \cdot{:} \lozenge\ )$
DEF $\widetilde{app}(s,\ a,\ \Gamma) = \widetilde{app}(\ rt(s),\ a,\ \Gamma \circ (\ ft(s) \cdot{:} \_\ )\ )$

Wie schon in Abschnitt 1.4.2 diskutiert, sollte man auf die erreichte Tail-Rekursion nicht allzu große Hoffnung setzen; denn hier wird lediglich der Keller in Form einer großen Funktionskomposition – einer so genannten *Closure* – explizit aufgebaut. (Compilertechnisch wird der Keller in den Heap verlagert.) Die *Continuation* bildet im Laufe der Zeit einen Funktionsterm der Bauart

$$id \circ (a_1 \cdot{:} \_\ ) \circ (a_2 \cdot{:} \_\ ) \circ \ldots \circ (a_n \cdot{:} \_\ )$$

der am Schluss auf die einelementige Liste $\langle a_{n+1} \rangle$ angewandt wird. Das heißt, Effizienz wird nicht gewonnen.

Versuch 2: Konkatenation

Der Operator „ $\cdot{:}$ “ ist nicht assoziativ. Aber er wird zum Spezialfall der Konkatenation, indem man $x \cdot{:} s$ durch $\langle x \rangle +\!\!+ s$ ersetzt. Damit sind die Assoziativitätsregeln aus Abschnitt 1.4 anwendbar und man erhält das folgende Definitionssystem:

DEF $app(s, a) = \widetilde{app}(s, a, \lozenge)$
DEF $\widetilde{app}(\lozenge,\ a,\ z) = z +\!\!+ \langle a \rangle$
DEF $\widetilde{app}(s,\ a,\ z) = \widetilde{app}(rt(s),\ a,\ z +\!\!+ \langle ft(s) \rangle)$

Hier ist zwar Tail-Rekursion erreicht, aber dafür haben wir die Konkatenationen $z +\!\!+ \ldots$ erhalten, die einen Aufwand von $\mathcal{O}(|z|)$ machen. Mit Hilfe der Techniken aus Abschnitt 12.2 kann das unter Umständen auf den amortisierten Aufwand $\mathcal{O}(2)$ reduziert werden, aber ein konstanter Overhead bleibt damit immer noch verbunden. Und die Verallgemeinerung auf komplexere Strukturen wie Bäume ist auch nicht offensichtlich.

Deshalb müssen wir nach einer weiteren Verbesserung suchen.

Versuch 3: Ein bisschen Seiteneffekt muss sein ...

Nachdem die ersten beiden Standardtechniken nicht so richtig geklappt haben, versuchen wir es mit einem allgemeinen Ansatz. Wie üblich suchen wir dazu nach einer geeigneten Einbettung. Deshalb betrachten wir die monadische Version von *app* aus Programm 13.5. (Schließlich müssen wir irgendwann auf dem Weg vom rein funktionalen Programm zum Maschinencode Seiteneffekte einführen.) Da die letzte – die „nachklappernde" – Operation $(x \cdot\!: s_2)$ am wichtigsten ist, setzen wir für sie ihre Definition aus Programm 13.4 ein.

$$
\begin{array}{lll}
\text{FUN } app\colon\ Seq \times \alpha \to Com\ Seq & & \\
\text{DEF } app(\Diamond, a) = new(a) & & \\
\text{DEF } app(s, a) = \ copy(s) \to s' & \& & \\
\qquad\qquad\qquad\quad\ \ ft(s) \to x & \& & \\
\qquad\qquad\qquad\quad\ \ rt(s') \to s_1 & \& & \\
\qquad\qquad\qquad\quad\ \ app(s_1, a) \to s_2 & \& & \\
\qquad\qquad\qquad\quad\ \ new(x) \to s_{new} & \& & -\!- \ ---- \\
\qquad\qquad\qquad\quad\ \ attach(s_{new}, s_2) & \& & -\!- \ x \cdot\!: s_2 \\
\qquad\qquad\qquad\quad\ \ yield(s_{new}) & & -\!- \ ----
\end{array}
$$

Wie wir in Abschnitt 1.4 gesehen haben, liegt das Hauptproblem in den nachklappernden Operationen. Deshalb versuchen wir davon möglichst wenige zu haben. Der erste entscheidende Trick besteht also darin, dass wir die Operation *new* nach oben verschieben, über den rekursiven Aufruf hinweg. Das geht problemlos, solange die Scopes von x und s_{new} beachtet werden.

$$
\begin{array}{lll}
\text{FUN } app\colon\ Seq \times \alpha \to Com\ Seq & & \\
\text{DEF } app(\Diamond, a) = new(a) & & \\
\text{DEF } app(s, a) = \ copy(s) \to s' & \& & \\
\qquad\qquad\qquad\quad\ \ ft(s) \to x & \& & \\
\qquad\qquad\qquad\quad\ \ new(x) \to s_{new} & \& & -\!- \ verschoben \\
\qquad\qquad\qquad\quad\ \ rt(s') \to s_1 & \& & \\
\qquad\qquad\qquad\quad\ \ app(s_1, a) \to s_2 & \& & \\
\qquad\qquad\qquad\quad\ \ attach(s_{new}, s_2) & \& & \\
\qquad\qquad\qquad\quad\ \ yield(s_{new}) & &
\end{array}
$$

Jetzt kommt der zweite entscheidende Trick: Wir betten die Funktion *app* in eine neue Funktion $\widetilde{app}$ ein, die zwei weitere Parameter enthält und den Rumpf von *app* ab dem rekursiven Aufruf erfasst (also den Teil, der „nachklappert").

Anmerkung: Wie kommt man auf diese Einbettung? Diese Idee folgt einem generellen Muster (s. auch Abschnitt 1.4): Wir betrachten eine Funktion folgender Bauart:

$$\text{DEF } f(x) = pre \ \& \ f(arg) \ \& \ post[\bar{z}]$$

wobei $\bar{z}$ für diejenigen Namen steht, die in pre eingeführt und in post benutzt werden (wobei jede Applikation eigens zählt). Dann ist diese Funktion trivialerweise gleichwertig zu folgender Einbettung:

$\text{DEF } f(x) = pre \;\&\; \widetilde{f}(arg, \bar{z})$
$\text{DEF } \widetilde{f}(x, \bar{y}) = f(x) \;\&\; post[\bar{y}]$

Mit dieser Einbettung rechnet man dann weiter in der Hoffnung, eine Tail-rekursive Form von $\widetilde{f}$ zu erzielen.

Wenn wir das auf *app* anwenden, dann hat s_{new} die Eigenschaft, vor dem rekursiven Aufruf (also im *pre*-Teil) definiert und nach dem rekursiven Aufruf (also im *post*-Teil) benutzt zu werden. Und weil es zweimal benutzt wird, brauchen wir zwei Parameter in der Einbettung. Wir nennen sie i (für *initial*) und p (für *previous*):

$$\begin{array}{l}\text{FUN } \widetilde{app}\text{: } Seq \times \alpha \times Seq \times Seq \to Com\ Seq \\ \text{DEF } \widetilde{app}(s, a, i, p) = \begin{array}{ll} app(s, a) \to s_3 & \& \\ attach(p, s_3) & \& \\ yield(i) & \end{array}\end{array}$$

Jetzt setzen wir die Definition von *app* ein, wobei wir bei der zweiten Gleichung $yield(s_{new}) \to s_3$ weglassen können, indem wir im folgenden *attach* direkt s_{new} anstelle von s_3 setzen.

$$\begin{array}{l}\text{DEF } \widetilde{app}(\Diamond, a, i, p) = new(a) \to s_3 \;\&\; attach(p, s_3) \;\&\; yield(i) \\ \text{DEF } \widetilde{app}(s, a, i, p) = \begin{array}{lll} copy(s) \to s' & \&-- & ----- \\ ft(s) \to x & \&-- & \\ new(x) \to s_{new} & \&-- & \\ rt(s') \to s_1 & \&-- & app(s, a) \to s_3 \text{ (bzw. } s_{new}) \\ app(s_1, a) \to s_2 & \&-- & \\ attach(s_{new}, s_2) & \&-- & ----- \\ attach(p, s_{new}) & \& & \\ yield(i) & & \end{array}\end{array}$$

Der Terminierungsfall für $\Diamond$ ist damit erledigt, weshalb wir nur den anderen weiter betrachten. Hier verschieben wir auch $attach(p, s_{new})$ weiter nach oben.

$$\begin{array}{l}\text{DEF } \widetilde{app}(s, a, i, p) = \begin{array}{lll} copy(s) \to s' & \& & \\ ft(s) \to x & \& & \\ new(x) \to s_{new} & \& & \\ attach(p, s_{new}) & \& & -- \text{ verschoben} \\ rt(s') \to s_1 & \& & \\ app(s_1, a) \to s_2 & \& & \\ attach(s_{new}, s_2) & \& & \\ yield(i) & & \end{array}\end{array}$$

Auf die letzten drei Zeilen passt jetzt wieder die Definition von $\widetilde{app}$, so dass wir folgende endgültige Definition erhalten:

$$\begin{array}{lll}
\text{DEF } \widetilde{app}(s,a,i,p) = & copy(s) \to s' & \& \\
& ft(s) \to x & \& \\
& new(x) \to s_{new} & \& \\
& attach(p, s_{new}) & \& \\
& rt(s') \to s_1 & \& \\
& \widetilde{app}(s_1, a, i, s_{new}) &
\end{array}$$

Diese Funktion ist tatsächlich *Tail-rekursiv*!

Als Letztes müssen wir noch klären, wie sich $\widetilde{app}$ zur ursprünglichen Funktion *app* verhält. Denn schließlich wird im umgebenden Programm ja *app* aufgerufen und nicht $\widetilde{app}$. Aber das ist einfach, weil die letzten drei Zeilen der Definition von *app* ja die Blaupause für die Definition von $\widetilde{app}$ lieferten. Entsprechend der obigen Musterfunktion f muss im Wesentlichen der *pre*-Teil erhalten werden gefolgt von einem Aufruf der neuen Funktion $\widetilde{f}$. Angewandt auf *app* liefert diese Überlegung folgende Form:

$$\begin{array}{lll}
\text{DEF } app(\Diamond, a) = new(a) & & \\
\text{DEF } app(s,a) = & copy(s) \to s' & \& \\
& ft(s) \to x & \& \\
& new(x) \to s_{new} & \& \\
& rt(s') \to s_1 & \& \\
& \widetilde{app}(s_1, a, s_{new}, s_{new}) &
\end{array}$$

Das zeigt, dass wir einen kleinen „Vorlauf“ brauchen, um die initialen Werte der neuen Parameter i und p zu bestimmen.

Anmerkung: Diese formale Rechnung sieht ja ganz interessant aus und es ist auch gut zu wissen, dass das Programm $\widetilde{app}$ und sein initialer Aufruf aus der Funktion app heraus garantiert korrekt sind, aber was ist eigentlich die Intuition hinter dieser Umformung? Auf Maschinenebene passiert hier Folgendes: Bei $new(x)$ wird im Heap eine Zelle erzeugt, die bildlich meistens folgendermaßen dargestellt wird:

Das Problem ist der Pointer. Er muss auf die Folgeliste zeigen, die aber erst während des rekursiven Aufrufs von app erzeugt wird. An dieser Stelle setzt der „Trick“ an: Wir generieren die Zelle trotzdem mit dem null-Pointer. Dann geben wir dem rekursiven Aufruf von app einen Zeiger auf die Zelle mit. (Das ist der Parameter p von $\widetilde{app}$.) Sobald im Rumpf die Nachfolgezelle generiert ist, wird ihr Pointer in p nachgetragen. Bei genauerer Analyse zeigt sich aber, dass man ganz am Schluss den Pointer auf die allererste Zelle abliefern muss; deshalb braucht man den weiteren Parameter i.

Diese Idee direkt in korrekten Maschinencode umzusetzen ist nichttrivial.[2] *Deshalb ist unsere obige Rechnung so wichtig: Sie stellt sicher, dass alle fummeligen Details richtig sind – beweisbar!*

[2] Vor einigen Jahren war das noch schwer genug, um Teil einer Doktorarbeit zu sein [128].

13.2.5 *Reference-Counting* und Queues: Hilft Laziness?

Wir sollten noch prüfen, ob die oben skizzierte Methode tatsächlich auch bei unseren eingangs vorgestellten Queue-Implementierungen hilft, d. h., Sicherheit mit akzeptabler Effizienz verbindet. Dabei werden wir zwei Dinge feststellen:

- Das Verfahren funktioniert auch in diesen Fällen einwandfrei.
- Es gibt einen Rest von Ineffizienz. Diesen kann man theoretisch zwar durch Laziness vermeiden, aber in der Praxis bringt das Nichts.

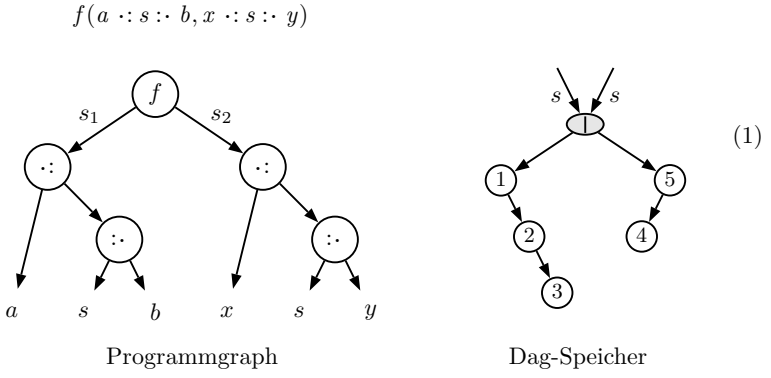
Betrachten wir dazu die einfachere Queue-Variante aus Abschnitt 12.2.1, bei der eine Queue nur ein Paar von Listen ist (vgl. Abbildung 12.1).

Beispiel 13.3 (Dag-Modell bei Queues)

Wir betrachten das kleine Programmfragment

$$f(\, a \cdot\!: s :\!\cdot\, b,\; x \cdot\!: s :\!\cdot\, y\,) \quad \text{WHERE } s = \langle 1,2,3,4,5\rangle$$

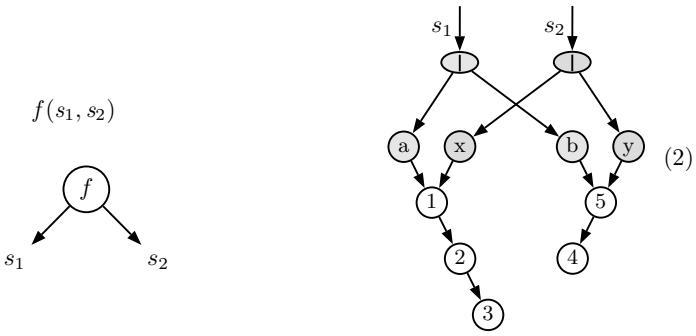
(1) Wir betrachten wieder die Situation in unserem Dag-Modell. Dieser Ausdruck hat einen Programmgraphen, in dem zwei Referenzen auf s vorkommen. Folglich hat der Dag zwei s-Kanten. Diese Kanten führen bei unserer Queue-Darstellung allerdings nicht direkt auf die Listen, sondern auf den Knoten, der die beiden Listenteile kontrolliert.



Programmgraph Dag-Speicher

Jetzt wird z. B. zuerst das linke Argument ausgewertet. Dabei werden – entsprechend unserer Implementierung in Abschnitt 12.2.1 – nur je eine Konstruktoroperation „ ·: “ auf die beiden Teillisten angewandt. Diese führen dazu, dass dann auch ein neuer Verwaltungsknoten entsteht. Der gleiche Prozess findet dann beim rechten Argument statt. (Dabei wird übrigens aufgrund der Single-Threadedness von s der Verwaltungsknoten wiederverwendet, der Rest der Liste aber nicht, weil dieser nach wie vor multi-threaded ist.)

(2) Nachdem die Argumente ausgewertet sind, liegt somit folgende Situation vor:



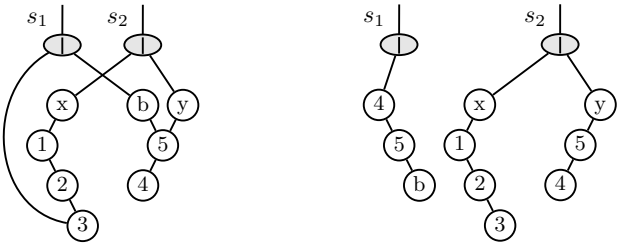
Programmgraph Dag-Speicher

Man sieht, dass in der Tat auch bei Queues keine unnötigen Kopieraktionen stattfinden – sogar dann, wenn die Queue multi-threaded ist.

Aber es gibt noch ein großes Problem! Wenn die rechte Teilliste – wie im obigen Beispiel – mehrfach benutzt wird, kann es zu Effizienzverlusten kommen, wenn sie revertiert werden muss. Das wollen wir uns im Folgenden genauer ansehen.

Beispiel 13.4 (Lazy Revert)

Wir setzen unser obiges Beispiel fort und nehmen an, dass wir uns ein Stück weit in die linke Liste s_1 hinabgearbeitet haben, so dass das Dag-Bild (a) auf der linken Seite vorliegt.



(a) vor $rt(s_1)$ (b) nach $rt(s_1)$

Wenn wir jetzt noch einmal $rt(s_1)$ ausführen, wird die Operation *revert* auf die *Tail*-Liste angewandt und es entsteht die Konfiguration (b) auf der rechten Seite.

Wenn wir jetzt auch die andere Liste s_2 hinabsteigen, muss nach einiger Zeit ebenfalls ein *revert* erfolgen, was für die Teilliste $\langle 5, 4\rangle$ Doppelarbeit bedeutet.

Aufgrund dieser Beobachtung liegt die Überlegung nahe, das *revert* mittels Laziness zu verzögern (wie es in Abschnitt 12.2.2 für die Queue-Implementierung bereits skizziert wurde). Dazu gibt es allerdings zwei Sichtweisen:

- *Pessimistische Sicht*: Theoretisch funktioniert das auch sehr schön, aber in der Praxis geht es doch sehr langsam. Denn die Operation *revert* muss ja auf jedem Element der Liste lazy sein. (Die Doppelarbeit beginnt z. B. in unserem Fall erst beim Element ⑤.) Und Laziness bedingt so viel Overhead, dass es billiger wird, die Liste ein zweites Mal zu revertieren.
- *Optimistische Sicht*: Etwas anders sieht die Sache dagegen aus, wenn man nicht die Laziness der Sprache (so sie Laziness bereitstellt) benutzt, sondern das Prinzip direkt in das Dag-Modell einbaut. Dann würde – in unserem obigen Beispiel – das *revert* bei multi-threaded Knoten wie ⑤ einen „Indirektionsknoten“ einbauen. Wenn dann die zweite *revert*-Operation auf einen solchen Knoten stößt, übernimmt sie die Restliste (d. h. nur noch die Kante) unmittelbar.

Fazit

Der Aufwand dieser Implementierungen sieht recht gewaltig aus, und er erreicht nach wie vor nicht die Effizienz der imperativen Standardimplementierung, die wir in Abbildung 13.1 am Anfang dieses Kapitels illustriert haben. Aber dazu muss man sich zwei Dinge vor Augen halten:

1. Die hier sichtbare intellektuelle Komplexität muss nur einmal beherrscht werden, und zwar vom Compilerbauer. Die Programmierer bleiben davon unbehelligt.
2. Das Problem entsteht durch die Notwendigkeit, persistente Queues sicher zu verwalten. Wenn eine Queue aber tatsächlich mehrfach gebraucht wird, dann muss sich auch der imperative Programmierer etwas anderes als Abbildung 13.1 einfallen lassen. (Und weil ihm das hier Diskutierte zu komplex sein wird, wird er im Allgemeinen auf banales Kopieren ausweichen – also schlechter sein als die standardmäßige funktionale Lösung.)

Bei Situationen, die single-threaded sind, wird der funktionale Compiler dagegen im Allgemeinen schlechter abschneiden, und zwar aus zwei Gründen:

- Eine Implementierung im Stil von Abbildung 13.1 könnte man zwar in funktionale Compiler einbauen, aber
  - erstens macht es viel Aufwand, zwei verschiedene Implementierungstechniken für den single- und den multi-threaded Fall einzubauen und konsistent miteinander in Einklang zu bringen;
  - und zweitens kann die Single-threaded-Analyse nicht alle Situationen erkennen.

- Deshalb verwenden die meisten Compiler – wenn überhaupt – das von uns hier skizzierte Verfahren (oder etwas Vergleichbares) sowohl für den single- als auch für den multi-threaded Fall. Wie wir gesehen haben, bleibt dabei ein Rest von Overhead erhalten (mit dem man aber gut leben kann, wie z. B. die Erfahrung mit dem OPAL-Compiler zeigt).

13.3 *Version Arrays*

Es bleibt noch die Frage zu diskutieren, wie es mit anderen wichtigen Strukturen aussieht, insbesondere mit *Array-artigen* Strukturen (die wir im Detail im folgenden Kapitel 14 studieren werden). In Abschnitt 13.1 hatten wir kurz den Typ *Matrix* angesehen, der ein Spezialfall von Arrays ist:

TYPE *Matrix* α
FUN *upd*: $\mathit{Matrix}\ \alpha \times \mathit{Index} \times \alpha \to \mathit{Matrix}\ \alpha$
FUN *sel*: $\mathit{Matrix}\ \alpha \times \mathit{Index} \to \alpha$

Hier stellt sich natürlich die gleiche Frage wie bei Listen. Wie gehen wir z. B. mit einer Situation der Art

$$\ldots f(\, \mathit{upd}(M, i, x),\ \mathit{upd}(M, j, y)\,) \ldots$$

um? Hier müssen wir zumindest eine der beiden Matrizen kopieren – also im Allgemeinen einen riesigen Aufwand leisten –, obwohl sie sich nur an einer Stelle von M unterscheidet. (Und bei der anderen können wir das Kopieren nur verhindern, wenn wir in unserem Compiler eine funktionierende und mächtige Single-threaded-Analyse besitzen.)

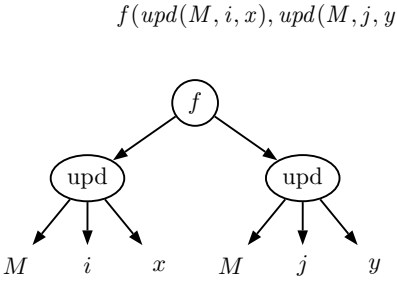
Deshalb versucht man auch hier einen Kompromiss zwischen garantierter Sicherheit, eingeschränkter Mächtigkeit von Single-threaded-Analysen und dem Wunsch nach Effizienz zu finden. Die Lösung folgt im Prinzip dem gleichen Ansatz wie das *Reference Counting* bei Listen, weshalb wir es im Folgenden auch ähnlich darstellen.

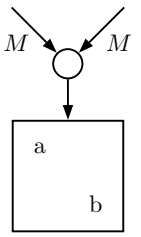
Beispiel 13.5

Wir betrachten den obigen Ausdruck $f(\, \mathit{upd}(M, i, x),\ \mathit{upd}(M, j, y)\,)$.

(1) In der Ausgangssituation haben wir zwei Referenzen auf M im Programmgraph und deshalb auch zwei M-Kanten im Speicher. Man beachte,

dass wir bei der Implementierung von Arrays einen Knoten als Indirektionsstufe eingebaut haben.



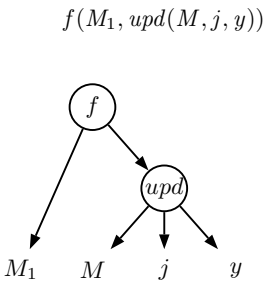


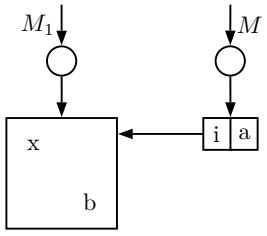
(1)

Programmgraph Dag-Speicher

(2) Wenn jetzt z. B. der linke *upd*-Ausdruck ausgewertet wird, dann wird der Array tatsächlich überschrieben. Außerdem wird ein neuer Indirektionsknoten auf den Array erzeugt und die Kante M_1 auf diesen Knoten als Ergebnis abgeliefert.

Aber außerdem wird auch noch der Inhalt des alten Indirektionsknotens geändert: Seine Ausgangskante zeigt jetzt auf eine neue Zelle, in der das Paar (i, a) vermerkt wird, also der alte Inhalt des Arrays an der Stelle i. Damit steht allen anderen Stellen im Programm noch die Ursprungsinformation zur Verfügung.

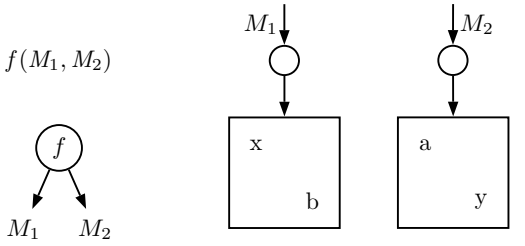




(2)

Programmgraph Dag-Speicher

(3) Wenn jetzt der zweite *upd*-Ausdruck ausgewertet wird, dann zeigt sich, dass die Matrix in der Tat in mehreren unterschiedlichen Varianten gebraucht wird. Deshalb wird jetzt auch eine Kopie erzeugt.



(3)

Programmgraph Dag-Speicher

Man beachte, dass bei diesem Verfahren kein *Reference counting* stattfindet. Der gleiche Effekt wird hier durch das Einschieben des Zwischenknotens ohnehin schon erreicht.

Natürlich könnte man hier das Kopieren noch weiter verzögern, indem man bei Bedarf weitere „Memory-Zellen“ vorschaltet. Aber damit werden die Zugriffe auf den Array immer langsamer, was seiner Natur widerspricht. Deshalb muss, sobald eine echte inkompatible Zweitverwendung auftritt, sofort kopiert werden.

Das hier gezeigte Verfahren mit den so genannten ***Version Arrays*** garantiert in jedem Fall Sicherheit und hinreichend schnelle Zugriffe, und vermeidet bei Single-Threadedness sogar unnützen Kopieraufwand. Deshalb finden sich Implementierungen solcher *Version Arrays* in mehreren funktionale Sprachen.

Wenn man trotz allem möchte, kann man natürlich auch weitere Memory-Zellen verwenden (sogar ganze „Memory-Arrays“). Aber dazu reichen dann die Programmiermittel der funktionalen Sprache selbst wieder aus, ohne dass man etwas am Compiler ändern muss.

14

Funktionale Arrays und Numerische Mathematik

Vielleicht werden wir uns dessen
einmal mit Freuden erinnern.
Vergil (Äneis)

Traditionell sind Arrays vor allem ein Zugeständnis an die Architektur der von Neumann-Rechner: Sie repräsentieren einen konsekutiven Speicherbereich, dessen Elemente sehr effizient und kompakt über Adressrechnung verarbeitbar sind. Manche Prozessoren stellen dafür sogar spezielle Instruktionen bereit. Die programmiersprachliche Abstraktion kaschiert diese Hommage an die Speicherarchitektur nur minimal.

Aber Arrays können wesentlich abstrakter gesehen werden. Auf semantischer und methodischer Ebene sind sie nichts anderes als spezielle Funktionen. Die effiziente Abbildung auf von Neumann-artige Speicher ist die Aufgabe des Compilers. Der kann dies – mit ein bisschen Hilfe – ohnehin besser. Damit nimmt man eine Sichtweise auf Arrays ein, die drei Charakteristika hat:

- Arrays sind *spezielle Funktionen*. Sie haben die beiden folgenden Besonderheiten, von denen die erste semantisch ist, die zweite pragmatisch.
- Der *Definitionsbereich* eines Arrays ist aus *Intervallen* der ganzen Zahlen gebildet. (Der Wertebereich ist beliebig.)
- Arrays sind *„eingefrorene" Funktionen*; d. h., sie liegen jeweils in einer tabellarischen Auflistung ihrer Werte vor. (Man kann das als eine Art genereller *Memoization*[1] auffassen.)

Man beachte: Der letzte Aspekt ist eine reine Erwartungshaltung an die Optimierungsfähigkeiten des Compilers. Er betrifft nicht das Arbeiten mit Arrays und vor allem nicht die Sicht von Arrays als speziellen Funktionen.

In den folgenden Abschnitten verfolgen wir noch ein wesentliches *methodisches* Ziel: Wir wollen das Arbeiten mit Arrays auf ein höheres und abstrakteres Niveau heben, als das mit der traditionellen Indexrechnung möglich ist.

[1] Der Begriff *Memoization* wird benutzt, wenn wiederholte Berechnungen von Funktionswerten durch Speicherung vermieden werden (s. Abschnitt 14.2.1).

Wir können das in Analogie zur Numerischen Mathematik sehen: Dort werden Matrixalgorithmen zuerst auf hohem Niveau präsentiert, bevor sie dann – im Vorgriff auf FORTRAN-Bedürfnisse – in Indexalgorithmen umgeschrieben werden. Auch bei den Programmiersprachen gibt es entsprechende Ansätze schon seit langem; eine der ersten Sprachen mit kompakten Vektor- und Matrix-Operatoren war APL. Aber auch moderne FORTRAN-Versionen bieten inzwischen ähnliche Schreibweisen. Im Bereich der Funktionalen Programmierung wurde das Problem z. B. in der Sprache SAC („Single-Assignment-C“) aufgegriffen [5, 127]. Im Zusammenhang mit der Programmierung von Parallelrechnern wurde der Bedarf an kompakteren Schreibweisen besonders offensichtlich (s. Kapitel 21).

Das alles passt gut zu einem generellen Grundprinzip der funktionalen Programmierung: Es gibt *keine selektiven Änderungen* von Datenstrukturen. Eine Funktion wie $update(A, i, x)$ ist *keine Änderung* des Arrays A an der Stelle i, sondern liefert einen *komplett neuen Array* A', der mit A überall, außer an der Stelle i übereinstimmt. Ob das auf Maschinenebene tatsächlich zum Kopieren führt, oder ob es doch mit punktuellem Überschreiben geht, ist eine Entscheidung, die wir der Optimierungskunst des Compilers überlassen. Wir kommen darauf im weiteren Verlauf des Kapitels nochmals zurück.

14.1 Semantik von Arrays: Funktionen

Wir fassen Arrays als spezielle Funktionen auf, deren Definitionsbereiche jeweils über Intervallen der ganzen Zahlen gebildet sind. Deshalb führen wir zunächst eine entsprechende Typklasse für diese Intervalle ein, auf der wir dann die Struktur der Arrays aufbauen.

14.1.1 Die Typklasse der Intervalle

Die Menge aller Intervalltypen wird in einer Typklasse zusammengefasst, die in Programm 14.1 spezifiziert ist. Die Typklasse der Intervalle besteht aus allen Subklassen von Int, die als Intervalle $(a..b)$ definiert sind. Für $a > b$ ist das Intervall leer; wir schreiben dafür auch $\emptyset$. (Man beachte, dass wir hier einen Typ in Mixfixnotation schreiben; das mag ungewohnt sein, ist aber weder technisch noch semantisch ein Problem.)

Mit den (polymorphen) Funktionen *first* und *last* erhält man jeweils Anfang und Ende eines Intervalls. Man beachte, dass diese beiden Funktionen als Argument einen Typ haben und als Resultat einen Wert dieses Typs. Deshalb braucht man abhängige Typen, um die Funktionalität adäquat auszudrücken.

Wir definieren auch die Vereinigung und den Durchschnitt zweier Intervalltypen, wobei darauf zu achten ist, dass das Ergebnis wieder ein Intervalltyp ist. Aus diesem Grund ist die Vereinigung nur definiert, wenn der Durchschnitt

Programm 14.1 Die Typklasse der Intervalle

```
SPECIFICATION Intervals = {
  TYPE _..._ : Int × Int → Interval                          -- Typ als Mixfix
  DEF a..b = ( Int | a ≤ _ ≤ b )
  FUN first, last: I: Interval ↦ I
  DEF first(a..b) = a
  DEF last(a..b) = b
  FUN _ ∩ _: Interval × Interval → Interval                  -- Durchschnitt
  DEF (a1..b1) ∩ (a2..b2) = (max(a1, a2)..min(b1, b2))
  FUN _ ∪ _: Interval × Interval → Interval                  -- Vereinigung
  DEF (a1..b1) ∪ (a2..b2) = IF max(a1, a2) < min(b1, b2)
                            THEN (min(a1, a2)..max(b1, b2)) FI
  FUN _ ⊆ _: Interval × Interval → Bool                      -- Subtyp
  DEF (a1..b1) ⊆ (a2..b2) = (a2 ≤ a1 ≤ b2) ∧ (a2 ≤ b1 ≤ b2)
  FUN Grow: Int → Interval → Interval                        -- vergrößern
  DEF Grow k (a..b) = IF k < 0 THEN (k + a..b)               -- k negativ!
                      IF k = 0 THEN (a..b)
                      IF k > 0 THEN (a..b + k) FI
  FUN Shift: Int → Interval → Interval                       -- verschieben
  DEF Shift k (a..b) = (a + k..b + k)                        -- I↑k bei positivem k
  FUN left: I ↦ I      VAR I: Interval                       -- I↓k bei negativem k
  FUN right: I ↦ I     VAR I: Interval
  DEF left i = (i − 1)
  DEF right i = (i + 1)
  FUN left: I × J ↦ I × J     VAR I: Interval, J: Interval
  FUN right: I × J ↦ I × J    VAR I: Interval, J: Interval
  FUN above: I × J ↦ I × J    VAR I: Interval, J: Interval
  FUN below: I × J ↦ I × J    VAR I: Interval, J: Interval
  DEF left (i, j) = (i − 1, j)
  DEF right (i, j) = (i + 1, j)
  DEF above (i, j) = (i, j − 1)
  DEF below (i, j) = (i, j + 1)
}
TYPECLASS Interval = Intervals.(_..._)
```

nicht leer ist oder die beiden Intervalle unmittelbar aneinander grenzen. (Andernfalls liegt ein Typfehler im Programm vor, der vom Compiler oder vom Laufzeitsystem gemeldet werden muss.)

Die Subtyp-Relation auf Intervallen lässt sich auf einfache Größenvergleiche der Grenzen zurückführen.

Mit der Operation *Grow* können wir Intervalle wachsen lassen. Ein negativer Wert von k verschiebt dabei die linke Grenze nach unten, ein positiver

Wert die rechte Grenze nach oben. Man beachte, dass *Grow* einen Typ liefert (weshalb wir es groß schreiben).

Mit *Shift* können wir das Intervall verschieben. Diese Operation ist sehr nützlich, weil viele mathematische Applikationen mit Indexshifts arbeiten. Als mnemotechnische Lesehilfe verwenden wir abhängig vom Vorzeichen von k die Pfeile $I^{\uparrow k}$ oder $I_{\downarrow k}$. Auch *Shift* liefert als Ergebnis einen Typ.

Auf den Intervalltypen definieren wir zur höheren Robustheit noch Operatoren *left*, *right*, *above* und *below*, die die Indexrechnung anschaulicher machen. Dabei unterscheiden wir – mittels Overloading – den eindimensionalen Fall (für Vektoren) und den zweidimensionalen Fall (für Matrizen). Man beachte, dass aufgrund der Typisierung implizit noch Typanpassungen wie z. B.

DEF $left[I]\ i = (i-1)\colon I$

erfolgen. Falls das Intervall über- oder unterschritten wird, ist daher das Ergebnis undefiniert (genauer: ein Typfehler). Aber aus Gründen der Lesbarkeit lassen wir möglichst oft die optionalen Parameter weg. Man beachte, dass alle diese Operationen polymorph sind.

Anmerkung: Hier zeigt sich die Mächtigkeit eines Systems, das Typen als „First-class citizens“ behandelt. Denn wir können mit den Definitionsbereichen von Arrays ganz normal rechnen, obwohl sie als Typen definiert sind.

14.1.2 Eindimensionale Arrays (Vektoren)

Eindimensionale Arrays sind Funktionen von Intervallen in beliebige Typen. Wir verwenden für diese Arrays auch den Begriff *Vektor*, obwohl dieser eigentlich nur auf Arrays mit numerischen Komponenten zutrifft.

Die Typklasse aller Arrays wird durch den Typkonstruktor $ArrayI\ \alpha$ generiert. Er hat zwei Argumente, den Definitionsbereich $I\colon \mathit{Interval}$, der ein Intervalltyp ist, und den Wertebereich $\alpha\colon \mathit{Type}$, der ein beliebiger Typ ist. Semantisch sind Arrays letztlich Funktionen, die den Indizes aus I Werte aus α zuordnen. Beispiel:

$$a\colon Array(0..2\cdot n)Real = \lambda\, i \bullet sin(\tfrac{i}{n}\pi)$$

Mit dieser Deklaration wird ein Array definiert, der $2n+1$ Elemente umfasst, die alle vom Typ *Real* sind. Die Zuordnung wird als λ-Ausdruck geschrieben.

Anmerkung: Manche Leute finden es irritierend, einen Array durch einen λ-Ausdruck zu definieren. Wenn das Gleiche in einer Spezialnotation wie z. B.

$$a = [\ i\colon (0..2\cdot n)\ \mid\ sin(\tfrac{i}{n}\pi)\ ]$$

geschrieben wird, fühlen sie sich erstaunlicherweise sofort wohler. Notationen dieser Bauart tragen – mathematischer Terminologie folgend – Namen wie Set Comprehension, List Comprehension oder Array Comprehension. Da für den Compiler solcher syntaktischer Schnickschnack keine Rolle spielt, sollten wir auch nicht unnötig Schreibweisen einführen. (In OPAL *gibt es einen Kompromiss: Arrays sind zwar spezielle Datenstrukturen mit entsprechenden Konstruktoren, aber man kann bei ihrer Generierung eine Funktion zur Initialisierung der Elemente angeben.)*

Programm 14.2 enthält die Definition der Typklasse der Arrays. Es spiegelt die Sicht wider, dass Arrays die Subklasse derjenigen Funktionen bilden, die als Definitionsbereich einen Intervalltyp haben. Damit ist insbesondere die Selektion $a(i)$ implizit verfügbar.

Programm 14.2 Die Typklasse der Arrays

```
SPECIFICATION Arrays = {
  TYPE Array: Interval → Type → Array                          -- Typkonstruktor
  DEF Array I α = (I → α)
      -- die Selektion a(i) ist damit implizit verfügbar
  FUN _ | _: Array I α × Ĵ: Interval ↦ Array (I ∩ J) α         -- Restriktion
  DEF a | J = λk: (I ∩ J) • a(k)
  FUN Dom: Array → Interval                                    -- Domaintyp
  FUN Ran: Array → Type                                        -- Rangetyp
  DEF Dom(Array I α) = I
  DEF Ran(Array I α) = α
  FUN _ ↦ _: î: Int × α ↦ Array (i..i) α                       -- Singleton
  DEF (i ↦ x) = λk: (i..i) • x
  FUN _ ⇐ _: Array I1 α × Array I2 α → Array I1 α              -- Überschreiben
        VAR I1: Interval, I2: Interval | I2 ⊆ I1
  DEF (a ⇐ b) = λk • IF k: (Dom b) THEN b(k) ELSE a(k) FI
  FUN _ _ ⇐ _: Array I α × I × α → Array I α                   -- Überschreiben
  DEF (a i ⇐ x) = (a ⇐ (i ↦ x))
  FUN _ ⊎ _: Array I1 α × Array I2 α → Array (I1 ∪ I2) α       -- Vereinigung
        VAR I1: Interval, I2: Interval, α: Type | I1 ∩ I2 = ∅
  DEF a ⊎ b = λk • IF k: (Dom a) THEN a(k)
                 IF k: (Dom b) THEN b(k) FI
  FUN _ ·: _: α × Array I α → Array (Grow(−1)I)α               -- prepend
  DEF x ·: a = (k ↦ x) ⊎ a WHERE k = first(Dom a) − 1
  FUN _ :· _: Array I α × α → Array (Grow(+1)I)α               -- append
  DEF a :· x = a ⊎ (k ↦ x) WHERE k = last(Dom a) + 1
}
TYPECLASS Array = Arrays.Array
```

Man beachte, dass wir in den (polymorphen) Funktionalitäten die Typvariablen nur dann explizit auflisten, wenn wir zusätzliche Restriktionen für sie angeben müssen. Ansonsten wird ihre Art aus der Verwendung im Typkonstruktor *Array* klar.

Wir überlagern hier den Namen *Array* der Typklasse und den Namen *Array* ihres Konstruktors, der ein polymorpher Typ ist. Diese Überlagerung sollte keine Probleme machen, weil aus dem Kontext immer klar ist, was jeweils gemeint ist; der unterschiedliche Font gibt noch eine weitere Lesehilfe.

Bei der Restriktion eines Arrays wird nur der Definitionsbereich (also der Typ) geändert; die Elemente bleiben unverändert.

Den Definitionsbereich eines Arrays (der ein Intervalltyp ist), erhalten wir über den Operator *Dom*. Analog liefert *Ran* den Wertebereich des Arrays.

Für einelementige Arrays verwenden wir die an die Mathematik angelehnte Schreibweise $(i \mapsto x)$. Außerdem nehmen wir uns die Freiheit, diese Notation auch auf mehrelementige Arrays auszudehnen und zu schreiben $(i_1 \mapsto x_1, \ldots, i_n \mapsto x_n)$.

Für Arrays erlauben wir *Updating*, das wir als Infixoperator schreiben. Die allgemeine Form $(a \Leftarrow b)$ überschreibt einen Array a partiell mit den Elementen eines Arrays b. Voraussetzung ist, dass der Indexbereich von b in dem von a enthalten ist. Für den Update an einer einzelnen Stelle sehen wir die Kurznotation $(a\ i \Leftarrow x)$ vor. Man beachte, dass diese Updates *nicht* eine selektive Änderung bedeuten (wie bei imperativen Programmiersprachen), sondern neue Arrays generieren. Allerdings lassen die Update-Operationen die Größe des Arrays unverändert; wenn Single-Threadedness gilt (vgl. Kapitel 13) kann der Compiler dies effizient durch Überschreiben implementieren.

Aber in vielen Situationen muss man auch neue Arrays aus existierenden ableiten, wobei die Größe sich ändert. Solche Operationen sind in imperativen Sprachen unbeliebt, weil sie fast immer auf ineffizientes Kopieren führen.[2] In funktionalen Sprachen sieht man das weniger problematisch, weil der Compiler ohnehin eine Menge Optimierungsarbeit leisten muss.

Die *Vereinigung* von zwei Arrays ist nur möglich, wenn ihre Indexbereiche passend sind: Sie müssen disjunkt sein und ihre Vereinigung muss wieder ein Intervalltyp sein. (Die zweite dieser Bedingungen wird in der Klasse *Interval* sichergestellt.)

Neben der Komposition ganzer Arrays brauchen wir auch spezielle Operationen für das Anhängen eines einzelnen Elementes vorne bzw. hinten an den Array. Diese Operationen sind das Gegenstück zu den Operationen *prepend* und *append* auf Sequenzen, weshalb wir die gleichen Symbole verwenden.

Man kann noch viele weitere nützliche Operationen einführen, z. B. die Konversion von Arrays zu Listen und umgekehrt. Aber wir beschränken uns auf die in Programm 14.2 angegebenen, mit denen wir alle folgenden Algorithmen formulieren können.

Zum Schluss sei noch ein Hinweis zur Notation gegeben. Wenn wir für die Definition von Arrays schon die λ-Notation

$$a = \lambda i \bullet sin(\tfrac{i}{n} \cdot \pi)$$

verwenden, dann können wir natürlich auch die Eleganz der musterbasierten Schreibweise übernehmen:

$$a(i) = sin(\tfrac{i}{n} \cdot \pi)$$

[2] In JAVA muss man deshalb z. B. von der eingebauten Struktur der Arrays zu der Bibliotheksklasse `Vector` übergehen; dabei kann man zur Effizienzsteigerung einen Schätzwert für die erwartete Maximalgröße angeben.

14.1.3 Mehrdimensionale Arrays (Matrizen)

In der Mathematik verwendet man eindimensionale Arrays, um Vektoren darzustellen. Aber es gibt auch *Matrizen*, für die man zweidimensionale Arrays braucht; allgemein hat man es sogar mit Matrizen beliebiger Dimension zu tun. Deshalb braucht man auch *mehrdimensionale Arrays*. Wir werden hier, ähnlich wie schon bei Vektoren, den Begriff Matrix auch für Arrays mit nichtnumerischen Elementen verwenden.

Da Arrays letztlich Funktionen sind, überträgt sich der bekannte mathematische Currying-Isomorphismus $A \times B \to C \simeq A \to B \to C$ auf Arrays:

$$Array\ (a_1..b_1) \times (a_2..b_2)\ \alpha \quad \simeq \quad Array\ (a_1..b_1)\ (Array\ (a_2..b_2)\ \alpha)$$

Aufgrund dieser Isomorphie verzichten manche Sprachen auf die Einführung von speziellen Notationen für mehrdimensionale Arrays (z. B. JAVA– auch wenn dort sicher nicht der Zusammenhang mit Currying die Motivation liefert). Wir wollen aber – je nach Situation – beide Notationen verwenden und führen deshalb eine entsprechende Variante des Typkonstruktors *Array* ein.

Programm 14.3 definiert die wesentlichen neuen Operatoren auf zweidimensionalen Arrays. (Höherdimensionale Arrays sind analog; wir beschränken uns hier auf den zweidimensionalen Fall.) Zur besseren Lesbarkeit verzichten wir wieder darauf, bei den polymorphen Funktionalitäten die Typvariablen explzit anzugeben.

Programm 14.3 Matrixförmige Arrays

```
TYPE Array : (Interval → Type) | (Interval × Interval → Type)
DEF Array I α = (I → α)
DEF Array (I × J) α = (I × J → α)

⋮  Selektion, Update, Singleton analog zu eindimensionalen Arrays

FUN trans: Array (I × J) α → Array (J × I) α          -- Transponieren
DEF trans a = λj, i • a(i, j)

FUN rows: Array (I × J) α = Array I (Array J α)       -- Currying (Zeilensicht)
DEF (rows a) = λi • (λj • a(i, j))

FUN cols: Array (I × J) α = Array J (Array I α)       -- Currying (Spaltensicht)
DEF (cols a) = λj • (λi • a(i, j))

FUN row: Array (I × J) α → I → Array J α              -- Zeile
DEF row a i = (rows a) i

FUN col: Array (I × J) α → J → Array I α              -- Spalte
DEF col a j = (cols a) j

FUN glue: Array I (Array J α) → Array (I × J) α       -- "Uncurrying"
DEF glue a = λi, j • a i j
```

Die Operation *trans* transponiert Zeilen und Spalten. Mit *rows* bzw. *cols* wird eine Matrix als Array von Zeilen bzw. als Array von Spalten aufgefasst. (Ersteres entspricht einem Currying, letzteres einem Currying mit vorausgehendem Transponieren.) Die beiden Operationen *row* und *col* sind nur Abkürzungen, die direkt eine Zeile bzw. Spalte auswählen. Beide liefern als Ergebnis also einen ganzen Vektor. Die Operation *glue* entspricht dem Uncurrying.

Wie sich zeigen wird, erlauben diese Operatoren einen sehr eleganten Programmierstil für Matrixprogramme. Dass sie auch effizient implementierbar sind, werden wir gleich in Abschnitt 14.2 sehen.

14.1.4 Matrizen von besonderer Gestalt (*Shapes*)

Der Einfachheit halber haben wir im vorigen Abschnitt erst einmal „normale“ Matrizen behandelt. Aber in der Mathematik treten an vielen Stellen Matrizen von spezieller Gestalt auf, vor allem Diagonal-, Tridiagonal-, obere und untere Dreiecksmatrizen usw. (s. Abbildung 14.1).

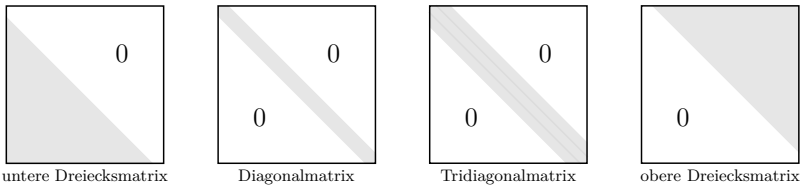


Abb. 14.1: Diagonal- und Dreiecksmatrizen

Bei der unteren und oberen Dreiecksmatrix nehmen wir jeweils die „echte“ Variante ohne die Diagonale. Varianten, die die Diagonale mit enthalten, wären ebenso leicht beschreibbar; wir erfassen sie aber lieber mit Hilfe des „ $\uplus$ “-Operators.

In der Mathematik sind die Komponenten von Matrizen Zahlen. Deshalb kann man diese Spezialmatrizen einfach dadurch ausdrücken, dass man den Rest jeweils mit Nullen auffüllt. Aber dies ist aus Effizienzgründen unerwünscht: Man will diese Nullen weder abspeichern noch verarbeiten. (Auch die Berechnung von $3 \cdot 0 \rightsquigarrow 0$ kostet Rechenzeit.)

Deshalb brauchen wir ein zusätzliches Konzept, nämlich die ***Gestalt*** (engl.: *Shape*) von Matrizen. Dies ist eine Erweiterung der Typinformation, die spezifiziert, welcher Teil des Indexbereichs $(I \times J)$ tatsächlich besetzt ist. Unser Ziel ist, dass Programme für solche Spezialmatrizen genauso geschrieben werden können wie Programme für allgemeine Matrizen (was das Programmieren vereinfacht). Die erwünschte Effizienz wird dann vom Compiler mit Hilfe dieser Typinformation automatisch generiert.

Im Wesentlichen sind Shapes Subtypen von $(I \times J)$ für gewisse Intervalle I und J. Wenn wir aber beliebige Subtypen zulassen, gelangen wir zum

allgemeinen Fall der *dünn besetzten Matrizen* (engl.: *sparse matrices*). Diese interessieren uns hier aber nicht; wir beschränken uns auf solche Matrizen, die immer noch eine kompakte Gestalt haben. Dazu fordern wir, dass die besetzten Komponenten jeder Zeile bzw. Spalte einen Vektor bilden.

Deshalb repräsentieren wir Shapes als Viertupel (I, J, R, C), die aus den beiden Grundintervallen und zwei Funktionen bestehen, die jeder Zeile bzw. Spalte das entsprechende besetzte Teilintervall zuordnen. Dies ist in Programm 14.4 skizziert.

Programm 14.4 Die Typklasse der Shapes

GROUP *Shapes* = {
  TYPECLASS $Shape = Shape(Interval, Interval, (Int \to Interval), (Int \to Interval))$
  TYPE $Diag[I] = Shape(I,\ I,\ \lambda i \bullet (i..i),\ \lambda j \bullet (j..j))$
  TYPE $TriDiag[I] = Shape(I, I,\ \lambda i \bullet (i-1..i+1) \cap I,\ \lambda j \bullet (j-1..j+1) \cap I)$
  TYPE $Lower[I] = Shape(I, I,\ \lambda i \bullet (first\ I..i-1),\ \lambda j \bullet (j+1..last\ I))$
  TYPE $Upper[I] = Shape(I, I,\ \lambda i \bullet (i+1..last\ I),\ \lambda j \bullet (first\ I..j-1))$
  TYPE $Rect[I, J] = Shape(I, J,\ \lambda i \bullet J,\ \lambda j \bullet I)$
  TYPE $(I \times J) = Rect(I, J)$
}

Aus Gründen der Lesbarkeit haben wir eine notwendige Bedingung hier nicht explizit aufgeschrieben: In einem Shape $S = (I, J, R, C)$ müssen die beiden Funktionen R und C *konsistent* sein. Das heißt, es muss gelten:[3]

$$\begin{aligned} \forall\ i{:}\, I,\ j{:}\, J \bullet\ & R(i) \subseteq J \\ & C(j) \subseteq I \\ & j{:}\, R(i) \iff i{:}\, C(j) \end{aligned}$$

Auf dieser Basis lassen sich dann die speziellen Gestalten von Diagonalmatrizen, Dreiecksmatrizen etc. sehr leicht definieren. Der ursprüngliche Fall der „normalen“ Matrizen lässt sich hier elegant durch den Shape $Rect[I, J]$ subsumieren. Wir definieren den Operator $(I \times J)$ entsprechend.

In Programm 14.5 redefinieren wir die wichtigsten Funktionen aus Programm 14.3 für den allgemeineren Fall der Shapes.

Für spezielle Matrizen kann man auch noch spezielle Operatoren einführen. So bieten sich z. B. für Diagonal- oder Tridiagonalmatrizen Operationen der folgenden Art an:

FUN _: $Array\ I\ \alpha \to Array\ (Diag\ I)\ \alpha$
FUN _: $Array\ I'\ \alpha \times Array\ I\ \alpha \times Array\ I''\ \alpha \to Array\ (TriDiag\ I)\ \alpha$

[3] Man könnte versuchen, nur eine der beiden Funktionen explizit anzugeben und die andere mit Hilfe dieser Konsistenzbedingung vom Compiler generieren zu lassen. Aber das geht im Allgemeinen selbst bei optimistischster Sicht über das hinaus, was man von Compilern erwarten darf.

Programm 14.5 Matrixförmige Arrays

```
TYPE Array Shape(I, J, R, C) α = (I × J → α)
⋮ Selektion, Update, Singleton analog zu eindimensionalen Arrays
FUN trans: Array (Shape(I, J, R, C)) α → Array (Shape(J, I, C, R)) α
DEF trans a = λj, i • a(i, j)
FUN rows: Array Shape(I, J, R, C) α → Array î: I (Array (C i) α)
DEF (rows a) = λ i • (λj • a(i, j))
FUN cols: Array Shape(I, J, R, C) α → Array ĵ: J (Array (R j) α)
DEF (cols a) = λj • (λ i • a(i, j))
FUN row: Array Shape(I, J, R, C) α → î: I → Array (C i) α
DEF row a i = (rows a) i
FUN col: Array Shape(I, J, R, C) α → ĵ: J → Array (R j) α
DEF col a j = (cols a) j
```

Hier werden die Matrizen über ihre Diagonalen definiert, die als Vektoren angegeben sind. (Bei Tridiagonalmatrizen müssen die beiden Nebendiagonalen entsprechend verkürzt sein.) Dabei schreiben wir die Generierung der Matrix jeweils wie einen unsichtbaren Konversionsoperator. Bei Bedarf verwenden wir auch entsprechende umgekehrte Typanpassungen, z. B. um Vektoroperationen auf Diagonalmatrizen anwenden zu können.

Die weiteren Funktionen wie z. B. die Vereinigung von Matrizen oder das Anhängen von Zeilen bzw. Spalten geben wir hier nicht explizit an. (Die notwendigen Prüfungen, ob die Indexbereiche disjunkt und die Ergebnisse wieder gültige Shapes sind, sind etwas umständlich aufzuschreiben.)

14.1.5 Map-Reduce auf Arrays

Wie auf allen gängigen Datenstrukturen kann man auch auf Arrays die Operatoren Map und Reduce definieren. Filter macht dagegen Probleme, weil im Allgemeinen keine Arrays mehr entstehen. Es wäre blanker Zufall, wenn die übrig gebliebenen Elemente gerade wieder Intervalle bildeten.

Anmerkung: Man kann einen Workaround basteln, indem man als Ergebnistyp von Filter Array I (Maybe α) nimmt. Dann kann man filtern, indem man die verschwundenen Elemente durch fail repräsentiert. Wir verfolgen diesen Ansatz hier nicht weiter.

Bei der Definition der Operatoren müssen wir allerdings eine schwierige *Designentscheidung* treffen. Was ist uns wichtiger:

- hohe Typsicherheit

oder

- kompakte und elegante Notationen?

Im ersten Fall müssen bei Matrixoperationen die Intervalle genau passen, während im zweiten Fall die Intervalle vom Compiler durch Adaption passend gemacht werden.

Wir entscheiden uns hier für die Eleganz. Es sei aber darauf hingewiesen, dass alle folgenden Programme sich auf die typsichere Variante umschreiben ließen; dabei müsste man nur immer wieder explizite Konversionsfunktionen einbauen. Programm 14.6 enthält Map und Reduce für eindimensionale Arrays. Mehrdimensionale Arrays sind analog.

Programm 14.6 Map und Reduce auf Arrays

```
FUN _*: (α → β) → Array I α → Array I β                    -- Map
DEF f * a = λ i • f(a(i))

FUN _ ⩛ _: Array I1 α × (α × β → γ) × Array I2 β → Array (I1 ∩ I2) γ   -- Zip
DEF a ⩛⊕ b = λ i • a(i) ⊕ b(i)

FUN _ / _: (α × α → α) × Array I α → α                     -- Reduce
DEF ⊕ /a = ⊕ / (a asList)

«analog auf mehrdimensionalen Arrays»
```

Der Map-Operator ist letztlich nichts anderes als die Funktionskomposition (weil Arrays Funktionen sind). Für den Zip-Operator verwenden wir die Mixfixnotation $a \overset{\oplus}{\curlyvee} b$. Er ist so definiert, dass Arrays mit verschiedenen Indexbereichen verbunden werden können. Das Ergebnis ist dann nur auf dem Durchschnitt der Indexbereiche definiert. Dieses Design ist entscheidend für die Eleganz vieler Algorithmen, verliert dafür aber etwas an Typsicherheit.

Anmerkung: Im Sinne von Kapitel 11 ist der Konstruktor Array I _ ein Funktor (analog zu Seq oder Set). Und der Map-Operator ist der Pfeil-Anteil dieses Funktors.

Den Reduce-Operator definiert man am besten über die Konversion in eine Liste. Damit ist die Ordnung der Elemente durch das Programm 14.2 geklärt. Wie üblich sollte man den Reduce-Operator ohnehin nur für assoziative und kommutative Operationen nehmen, da man sonst keinerlei Intuition über seinen Effekt hat. Das gilt bei Matrizen in noch stärkerem Maße als bei Vektoren.

14.2 Pragmatik von Arrays: „Eingefrorene“ Funktionen

Wenn Arrays einfach nur Funktionen wären, bräuchte man keine eigene Begrifflichkeit für sie. Es gibt aber eine zweite charakteristische Eigenschaft, die sie zu etwas Besonderem macht: *Arrays sind „eingefrorene“ Funktionen.* Diese Eigenschaft lässt sich auf der funktional-semantischen Ebene *nicht* ausdrücken; sie ist ein pragmatischer Aspekt, der im Compiler realisiert werden

muss. Abbildung 14.2 illustriert das Prinzip: Für einen Array mit Definitionsbereich $(a..b)$ werden im Speicher $b - a + 1$ konsekutive Zellen reserviert, in die die entsprechenden Arraywerte (also die Funktionswerte) eingetragen werden.

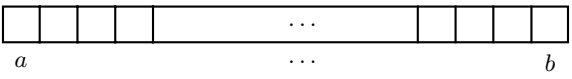


Abb. 14.2: Array als Speicherblock

Das ist allerdings eine etwas vereinfachende Sicht, die wir gleich noch weiter verfeinern müssen. Zuvor wollen wir uns aber mit der wichtigeren Frage befassen, wie das Verhältnis zwischen diesem implementierungstechnischen Begriff von Speicherblöcken und dem semantischen Funktionsbegriff aussieht.

14.2.1 Memoization

Das Prinzip „Speichern statt (neu) Berechnen“ ist in optimierenden Compilern unter dem Begriff ***Memoization*** bekannt: Sobald der Wert einer Funktion f an der Stelle i berechnet ist, wird er abgespeichert. Bei jedem weiteren Aufruf von $f(i)$ wird dann anstelle der (teuren) Neuberechnung der gespeicherte Wert genommen.

Bei der Verwendung von Memoization im Rahmen von Optimierung ist das zentrale Problem die Entscheidung, welche Werte mehrfach benötigt werden, sodass sich das Aufbewahren lohnt. Bei uns geht es darum, bei der Erzeugung des Arrays eine adäquate Berechnungsreihenfolge zu identifizieren. Betrachten wir als Einstieg noch einmal den Array

$$a\colon \mathit{Array}\ (0..2 \cdot n)\ \mathit{Real} = \lambda\, i \bullet \mathit{sin}(\tfrac{i}{n}\pi)$$

Hier ist das Problem trivial: Die Werte a_i können in beliebiger Reihenfolge ausgerechnet und abgespeichert werden. Der Compiler hat also alle Freiheiten.

Kritischer ist es, wenn die Elemente des Arrays rekursiv definiert werden. So kann z. B. der Array $(1, 2, 4, \ldots, 1024)$ folgendermaßen definiert werden:

$$a\colon \mathit{Array}\ (0..10)\ \mathit{Int} = \lambda\, i \bullet \textsc{if}\ i = 0\ \textsc{then}\ 1 \\ \textsc{if}\ i > 0\ \textsc{then}\ 2 \cdot a(i-1)\ \textsc{fi}$$

Aus semantischer Sicht ist diese Rekursion harmlos, da Arrays nur Funktionen sind. Deshalb sieht dieser Array a auch nicht anders aus als z. B. die Fakultätsfunktion. Das Problem liegt im „effizienten Einfrieren“. Um das zu verstehen betrachten wir die zwei „natürlichen“ Ordnungen, in denen der Speicher gefüllt werden kann.

- *Aufsteigend.* Der Compiler setzt der Reihe nach die Felder $0, 1, 2, \ldots, 10$. Betrachten wir einen Schnappschuss des Füllprozesses:

1	2	4	8	16						
0	1	2	3	4	5	6	7	8	9	10

Wenn in dieser Situation das Feld $a(5)$ besetzt wird, dann muss der Compiler dazu laut Definition den Wert $2 \cdot a(4)$ auswerten. Da das Feld $a(4)$ schon besetzt ist, erfordert das $\mathcal{O}(1)$ Zeit. Insgesamt wird der Array mit linearem Aufwand $\mathcal{O}(n)$ gesetzt.

- *Absteigend **ohne** Memoization.* Der Compiler setzt der Reihe nach die Felder 10, 9, 8, ..., 0. Betrachten wir einen entsprechenden Schnappschuss:

						64	128	256	512	1024
0	1	2	3	4	5	6	7	8	9	10

Wenn in dieser Situation das Feld $a(5)$ besetzt wird, dann muss der Compiler dazu den Wert $2 \cdot a(4)$ auswerten. Dazu wird rekursiv $a(3)$ aufgerufen, was wiederum den Aufruf $a(2)$ triggert usw. Insgesamt ergibt sich für den ganzen Array ein Aufwand in der Ordnung $\mathcal{O}(n^2)$.

- *Absteigend **mit** Memoization.* Der Compiler setzt der Reihe nach die Felder 10, 9, 8, ..., 0. Aber jetzt passiert Folgendes: Beim Setzen von $a(10)$ wird $a(9)$ aufgerufen, was $a(8)$ aufruft usw. Am Ende der Kette wird $a(0) = 1$ berechnet *und gespeichert*. Dann wird der Aufruf von $a(1)$ abgeschlossen *und das Ergebnis* 2 *gespeichert*. Und so weiter. Schließlich wird der Aufruf $a(9)$ beendet und das Ergebnis 512 gespeichert. Damit wird dann $a(10) = 1024$ beendet und gespeichert.
  Das heißt: Am Ende der ersten Setzung $a(10)$ *ist als „Seiteneffekt“ der ganze Array gesetzt*. Danach initiiert der Compiler die Setzung von $a(9)$; weil das aber schon gesetzt ist, ist nichts mehr zu tun. Und so weiter.

Dieses Beispiel illustriert, dass mittels Memoization der Aufwand immer in der Ordnung $\mathcal{O}(n)$ liegt, allerdings wegen des höheren Verwaltungsaufwands mit einem größeren konstanten Faktor als bei der optimalen Reihenfolge. Deshalb wird die Entscheidung, in welcher Reihenfolge ein Array gefüllt wird, zu einem wichtigen Effizienzkriterium.

Festlegung (Memoization)

Wir gehen davon aus, dass der Compiler beim „Füllen“ von Arrays (die als λ-Ausdrücke definiert sind) Memoization verwendet. In den meisten Fällen wird der Optimierer sogar in der Lage sein, die optimale Füllrichtung zu erkennen, so dass auch die Prüfung auf „schon da?“ entfallen kann.

Anmerkung: Um allgemeines Memoization zu realisieren, gibt es im Wesentlichen zwei Möglichkeiten. Entweder man geht compilerintern für die Arrayelemente auf den Typ Maybe[α] über. Oder man verwendet einen gleich großen Bit-Array, der angibt, welche Elemente im Hauptarray schon da sind.

Falls wir das Optimierungsproblem als zu komplex für den Compiler erachten, können wir als *Pragmatik* auch noch entsprechende Operatoren auf den Intervalltypen einführen. Wir schreiben

- „$a..b$“ für einen Intervalltyp, bei dem wir dem Compiler keine Ordnungshinweise geben,
- „$a \nearrow b$“ für einen Intervalltyp, bei dem wir dem Compiler eine aufsteigende Ordnung empfehlen, und
- „$a \swarrow b$“ für einen Intervalltyp, bei dem wir dem Compiler eine absteigende Ordnung empfehlen.

Man beachte: Dies sind reine Hinweise an den Compiler; sie ändern nichts an der Semantik von Arrays als Funktionen.

Es gibt noch eine Komplikation, die wir beachten müssen. Wir stoßen manchmal auf Arrays, die verschränkt rekursiv voneinander abhängen (ein Beispiel werden wir bei der Gauß-Elimination antreffen):

$$A = \lambda k \bullet \text{«}\ldots A \ldots B \ldots\text{»}$$
$$B = \lambda k \bullet \text{«}\ldots A \ldots B \ldots\text{»}$$

Zur Berechnung der Werte $A(k)$ brauchen wir Werte von B und umgekehrt. Egal welchen Array wir zuerst besetzen, wir benötigen für den jeweils anderen Memoization. Um das nach Möglichkeit zu vermeiden, besetzt man beide Arrays *simultan*: Man wählt eine Reihenfolge für die Indizes k (aufsteigend oder absteigend) und berechnet für jeden Index sowohl das Element $A(k)$ als auch das Element $B(k)$. *Diese simultane Berechnung wird bei verschränkt rekursiven Arrays vom Compiler grundsätzlich benutzt.*

14.2.2 Speicherblöcke

Dies ist kein Buch über Compilerbau. Aber man sollte trotzdem eine Vorstellung davon haben, wie die einzelnen Sprachkonzepte in der Maschine umgesetzt werden. (Auch ein Architekt sollte zumindest grobe Vorstellungen von Statik haben, um zu wissen, ob sein Haus stehen bleibt, und um zu ahnen, was es ungefähr kosten wird.)

Bei der Implementierung von Arrays müssen wir, wie üblich, eine Abwägung zwischen Effizienz und Sicherheit vornehmen. Dabei legen funktionale Sprachen großen Wert auf Sicherheit und spendieren dafür etwas Rechenzeit und Speicherplatz.

Anmerkung: Bei imperativen Sprachen ist es umgekehrt: Man legt großen Wert auf Effizienz und nimmt dafür eine größere Fehleranfälligkeit in Kauf. Spötter behaupten, dass Sprachen wie JAVA *oder* .NET *Effizienz geopfert haben, ohne wesentlich an Sicherheit zu gewinnen.*

14.2.3 Sicherheit und Single-Threadedness: Version Arrays

Die erhöhte Sicherheit wird erreicht, indem versehentliches Überschreiben noch benötigter Werte unterbunden wird. Die dazu notwendigen Techniken wurden generell schon in Kapitel 12 im Zusammenhang mit Single-Threadedness besprochen.

Wegen der Größe von Arrays ist es dabei besonders wichtig, dass man sie nicht unnötig kopiert. Die entsprechenden Techniken haben wir unter dem Stichwort *Version Arrays* schon in Kapitel 13 in Abschnitt 13.3 kennen gelernt.

14.2.4 Von Arrays zu Speicherblöcken

Wir wenden uns nun der generellen Frage der effizienten Speicherung von Arrays zu. Die einschlägigen Techniken sind aus dem Compilerbau bekannt; wir beschränken uns hier auf eine knappe Skizze der prinzipiellen Konzepte.

Für die effiziente Verarbeitung von Arrays der Art $Array(a..b)\alpha$ benutzt der Compiler Speicherblöcke mit $n = b - a + 1$ Elementen vom Typ α, die von $0..n-1$ indiziert sind (s. Abbildung 14.3).[4]

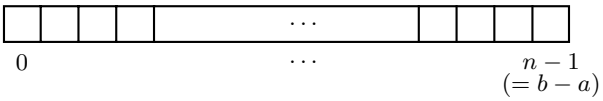


Abb. 14.3: Array als Speicherblock

Dazu wird vom Compiler eine intern vorgegebene Struktur *Block* benutzt. Diese dient – ähnlich wie der Dag bei den Heaps aus Kapitel 12 – als Hidden state einer entsprechenden Monade.

```
STRUCTURE Blocks = {                          -- vorgegeben vom Compiler
  TYPE Block[n: Nat]α
  FUN empty: Block[n]α
  DEF empty = «n undefinierte Werte vom Typ α»
  FUN get: Block[n]α × Nat → α
  DEF get(block, i) = «i-tes Element»
  FUN set: Block[n]α × Nat × α → Block[n]α
  DEF set(block, i, a) = «i-tes Element auf den Wert a setzen»
  FUN step: Nat → Nat                         -- Schrittfunktion für iterate
  FUN iterate: ...                            -- monadisch
       ...
}
```

[4] In Sprachen wie JAVA wird dem Programmierer überhaupt nur diese Primitivform von Arrays zur Verfügung gestellt. Alles andere muss man selbst implementieren.

Dieser einfache Basistyp ist dann noch um weitere Typinformation über den jeweiligen Array zu ergänzen, die im Compilerbau üblicherweise als ***Array-Deskriptoren*** bezeichnet werden. (Da wir generell mit dynamischen Typen arbeiten, können wir sie als Teil der Typinformation behandeln.)

14.2.5 Implementierung eindimensionaler Arrays

Wir betrachten zunächst den einfachen Fall eines eindimensionalen Arrays. Tabelle 14.1 skizziert die wesentliche Essenz der Übersetzung der Array-Operationen in entsprechende Block-Operationen. Diese Operationen werden im Compiler de facto monadisch realisiert, wobei der Block die Rolle des Hidden state übernimmt. (Dies ist analog zu Kapitel 13.)

Das Array-Konstrukt ...	... wird übersetzt in
$Array(a..b)\alpha$	$Block(n)\alpha$ WHERE $n = b - a + 1$
$v(i)$	$get(block, i - a)$
$v\ i \Leftarrow x$	$set(block, i - a, x)$
step	$(+1)$, (-1) etc.
$v = \lambda i \bullet f(v, i)$	$iterate(\ldots)$

Tab. 14.1: Übersetzung von Array-Operationen

- Der Typ $Array(a..b)\alpha$ wird in einen Block der Länge $b - a + 1$ konvertiert.
- Die Array-Selektion $v(i)$ wird auf die Blockselektion abgebildet, wobei ein Indexshift in das Intervall $0..n - 1$ vorgenommen wird. (Die Einhaltung der Indexgrenzen $a \leq i \leq b$ ist Teil der Typprüfung, die vom Compiler generell eingebaut wird und deshalb hier nicht noch einmal angegeben werden muss.)
- Der Array-Update wird entsprechend auf *set* abgebildet.
- Die Operation *step* kodiert die Richtung und Schrittweite, in der der Array verarbeitet werden soll (s. nächster Punkt).
- Die Erzeugung eines Arrays über einen Ausdruck, also $v = \lambda i \bullet f(v, i)$, wird sequenziell über einen Iterator gelöst, der mit Hilfe der Operation *step* die Arrayelemente nacheinander mit den entsprechenden Werten von $f(v, i)$ besetzt. Dabei muss man für *step* drei Fälle unterscheiden. (1) Wenn alle Zugriffe $v(j)$ im Ausdruck f Indizes $j < i$ haben, dann nimmt man $(+1)$; (2) falls alle $v(j)$ sich auf Indizes $j > i$ beziehen, dann nimmt man (-1); (3) falls beides vorkommt, muss man *Memoization* verwenden. (Bei Matrizen werden wir komplexere Formen von *step* erhalten.)

Man kann noch eine effiziente Version des Restriktionsoperators vorsehen. Denn bei einer Deklaration der Art

$u = (\, v \mid J \,)$

kann man oft darauf verzichten, den entsprechenden Ausschnitt des Arrays v zu kopieren. Stattdessen beschränkt man sich bei u auf eine Angabe des Teilintervalls J zusammen mit einer Referenz auf u. (Das Kopieren kann dann immer noch nötig werden, wenn die Bedingungen der Single-Threadedness nicht erfüllt sind; s. dazu Abschnitt 13.3.)

Damit ergibt sich der *Array-Deskriptor* als ein Tupel mit folgenden Komponenten: eine Referenz auf den Block, die Grenzen a und b, die Schrittfunktion *step* und gegebenenfalls das Teilintervall J.

14.2.6 Implementierung mehrdimensionaler Arrays

Bei mehrdimensionalen Arrays ergeben sich ähnliche Umsetzungen, die in Tabelle 14.2 angegeben sind. Dabei beschränken wir uns auf den Fall der normalen (also rechteckigen) zweidimensionalen Matrizen.

Das Array-Konstrukt ...	... wird übersetzt in
$Array(a_1..b_1) \times (a_2..b_2)\alpha$	$Block[n_1 \cdot n_2]\alpha$ WHERE $n_1 = b_1 - a_1 + 1,\ n_2 = b_2 - a_2 + 1$
$m(i,j)$	$get(block, i \cdot n_2 + j - s)$ WHERE $s = a_1 \cdot n_2 + a_2$
$m\ (i,j) \Leftarrow x$	$set(block, i \cdot n_2 + j - s,\ x)$ WHERE $s = a_1 \cdot n_2 + a_2$
step	$(+1)$, (-1) etc.
$v = \lambda i,j \bullet f(v,i,j)$	$iterate(\ldots)$

Tab. 14.2: Übersetzung von Matrixoperationen (einfache Variante)

Die Formel bei der Selektion und beim Update ergibt sich als leichte Optimierung der eigentlichen Formel $(i - a_1) \cdot n_2 + (j - a_2)$: Weil der Term $s = (a_1 \cdot n_2 + a_2)$ unabhängig von i und j ist, kann man ihn vorab berechnen.

Bei normalen Rechteckmatrizen kann man bei der Setzung mit einem einzigen Iterator arbeiten, der mit Hilfe von *step* alle Komponenten aufsteigend oder absteigend durchläuft. Aber sobald man allgemeinere Shapes zulässt oder Operatoren wie *trans*, *rows* oder *cols* effizient implementieren möchte, wird die Umsetzung komplexer. Insbesondere wollen wir Matrizen auch in transponierter Form oder in Zeilen- bzw. Spaltensicht definieren können. Damit entstehen z. B. Definitionen der folgenden Bauart:

$$\begin{aligned}
&\ldots \text{ WHERE } (\mathit{trans}\ a) &&= \lambda j, i \bullet c[\ldots a(i', j') \ldots] \ \ldots \\
&\ldots \text{ WHERE } (\mathit{rows}\ a) &&= \lambda i \bullet \lambda j \bullet c[\ldots a(i', j') \ldots] \ \ldots \\
&\ldots \text{ WHERE } (\mathit{cols}\ a) &&= \lambda j \bullet \lambda i \bullet c[\ldots a(i', j') \ldots] \ \ldots
\end{aligned}$$

Hier wird z. B. in der ersten Zeile eine Matrix a definiert, aber nicht direkt, sondern indem gesagt wird, wie ihre Transponierte aussieht. Diese Transponierte wird hier punktweise über einen geeigneten Ausdruck $c[\ldots]$ bestimmt. Dabei dürfen sogar (wie üblich) rekursive Verwendungen von Elementen $a(i', j')$ der gerade definierten Matrix a selbst vorkommen. Man beachte, dass diese Zugriffe sich auf die nicht-transponierte Form von a beziehen. Deshalb muss der *Array-Deskriptor* hier die gesamte Shape-Information $\mathit{Shape}(I, J, R, C)$ enthalten, und zwar zusammen mit der Information, ob eine transponierte Sicht, eine Zeilen- oder eine Spaltensicht eingenommen wird.

Wenn wir z. B. die dritte der obigen Definitionen betrachten, dann führt sie in optimierter Form auf einen geschachtelten Iterator der Bauart

$$\mathit{iterate}(\mathit{step}_1, \ldots, \mathit{iterate}(\mathit{step}_2, \ldots a(i, j) \Leftarrow c[\ldots a(i', j') \ldots])),$$

wobei $\mathit{step}_1(j) = j + 1$ und $\mathit{step}_2(i) = R(i)$. Das heißt, in der inneren Schleife springt man immer um die Länge der aktuellen Zeile weiter. Außerdem muss man noch berechnen, an welcher Stelle die jeweilige Iteration anfängt und wo sie aufhören muss.

Aber, wie schon gesagt, dies ist kein Buch über Compilerbau; deshalb arbeiten wir diese Details hier nicht weiter aus.

Mit dieser kurzen Skizze sollte nur plausibel gemacht werden, dass auch für die Matrizen mit speziellen Gestalten alle in den Programmen 14.2 bis 14.6 aufgeführten Operationen effizient implementierbar sind. Deshalb können wir im Folgenden bei der exemplarischen Entwicklung von Array-Algorithmen problemlos auf diesem Abstraktionsniveau arbeiten.

14.3 Arrays in der Numerik: Vektoren und Matrizen

Nach den allgemeinen Einführungen in das Konzept der Arrays konzentrieren wir uns jetzt auf die wichtigsten Spezialfälle, nämlich Vektoren und Matrizen. In beiden Fällen sind die Elemente numerische Typen, aber Vektoren sind eindimensionale und Matrizen zweidimensionale Arrays. (Matrizen höherer Dimension betrachten wir nicht; sie verhalten sich aber analog.)

Für diese Vektoren und Matrizen hat man in der Mathematik eine reichhaltige Palette von Operatoren, die wir auch für die Programmierung verfügbar haben sollten. Zum Glück lassen sie sich alle (effizient) auf unsere Basisoperatoren für Arrays zurückführen, insbesondere auf *Map* und *Reduce*.

14.3.1 Vektoren

Wir beginnen mit Vektoren, also eindimensionalen Arrays über Zahlen. Programm 14.7 enthält die Definition der entsprechenden Struktur. Um den

Schreibaufwand zu reduzieren, parametrisieren wir die ganze Struktur mit dem Typ der Arrayelemente. Wir können damit z. B. Vektoren über ganzen, reellen aber auch komplexen Zahlen verwenden.

Programm 14.7 Vektoren

```
STRUCTURE Vectors(Number: Numeric) = {
  TYPE Vector[I: Interval] = Array I Number
  TYPE Vector[n: Nat] = Array (1..n) Number
  FUN _ + _: Vector × Vector → Vector          -- Addition
  DEF u + v = u ⨁ v
  FUN _ − _: Vector × Vector → Vector          -- Subtraktion
  DEF u − v = u ⨀ v
  FUN _ · _: Vector × Vector → Number          -- Skalarprodukt
  DEF u · v = + / (u ⨁ v)
  FUN inv: Vector → Vector                     -- Inverse
  DEF inv v = (λx • 1/x) ∗ v                   -- geschrieben v^-1
  FUN _ · _: Number × Vector → Vector          -- Multiplikation mit Skalar
  DEF x · u = (x · _ ) ∗ u
  FUN _ · _: Vector × Number → Vector          -- Multiplikation mit Skalar
  DEF u · x = (x · _ ) ∗ u
}
```

Wir sehen zwei (überlagerte) Versionen des Typkonstruktors vor. Im allgemeinen Fall haben Arrays als Indexbereich ein beliebiges Intervall $(a..b)$. Aber meistens ist der Indexbereich gerade $(1..n)$; deshalb sehen wir für diesen Standardfall eine abkürzende Notation vor, bei der nur n anzugeben ist.

Für Vektoren definieren wir die Standardoperationen der (elementweisen) Addition und Subtraktion sowie das Skalarprodukt. Da wir – der Mathematik folgend – den Operator „ · “ für das Skalarprodukt verwenden, müssen wir die elementweise Multiplikation als „ $\dot{\bigvee}$ “ schreiben.

Man beachte, dass diese Operationen aufgrund der Definition des Zip-Operators auf dem *Durchschnitt* der Indexbereiche der beiden Argumentvektoren arbeiten. (Das ist zwar weniger typsicher, wird sich aber als sehr nützlich in einigen der späteren Applikationen erweisen.)

Die Inverse eines Vektors wird durch komponentenweise Inversenbildung realisiert.

Außerdem sehen wir die Multiplikation eines Vektors mit einem Skalar vor.

14.3.2 Matrizen

Bei Matrizen haben wir in Analogie zu Vektoren die punktweise Addition und Subtraktion sowie die Multiplikation mit einem Skalar. Dazu kommen dann die Operationen des Matrixprodukts sowie der Multiplikation einer Matrix mit einem Vektor.

Programm 14.8 Matrizen

```
STRUCTURE Matrices(Number: Numeric) = {
  TYPE Matrix[S: Shape] = Array S Number

    ⋮  Addition, Subtraktion, Multiplikation mit Skalar analog zu Vektoren

  FUN _ · _: Matrix(I × K) × Matrix(K × J) → Matrix(I × J)-- Matrixprodukt
  DEF a · b = λ i, j • a.row(i) · b.col(j)
  FUN _ · _: Vector J × Matrix (I × J) → Vector I
  DEF v · a = (v·) ∗ (cols a)
  FUN _ · _: Matrix (I × J) × Vector I → Vector J
  DEF a · v = (·v) ∗ (rows a)
}
```

Bei der Matrixmultiplikation wird die übliche „Zeile-mal-Spalte“-Definition direkt umgesetzt. Bei den beiden Formen des Vektor-Matrix-Produkts wird jeweils das Skalarprodukt mit allen Spalten bzw. allen Zeilen gebildet.

Aufgrund der Definitionen der Map- und Zip-Operationen auf Arrays sind die obigen Definitionen auch für die speziellen Gestalten der Diagonalmatrizen, Dreiecksmatrizen etc. effizient realisiert. Allerdings ist die exakte Definition der Typkorrektheit bei allgemeinen Shapes etwas aufwendig. Deshalb beschränken wir uns hier auf die Angabe für den Spezialfall der normalen Rechteckgestalt.

14.4 Beispiel: Gauß-Elimination

Ein klassisches Beispiel für Matrixrechnung ist die so genannte ***Gauß-Elimination***, mit der man lineare Gleichungssysteme lösen kann, die in Matrixform gegeben sind (mit einer gegebenen $n \times n$-Matrix A und einem gegebenen Vektor $\mathbf{b}$):

$$A \cdot \mathbf{x} = \mathbf{b} \tag{14.1}$$

Um das Gleichungssystem für verschiedene Eingabevektoren $\mathbf{b}_1, \ldots, \mathbf{b}_n$ lösen zu können, führt man für die Matrix A eine so genannte LU-Zerlegung durch.

$$A = L \cdot U \tag{14.2}$$

In den üblichen Darstellungen in der Literatur ist L eine untere und U eine obere Dreiecksmatrix, wobei die Diagonale von L mit 1 besetzt ist (vgl. Abbildung 14.4). Wir wählen hier eine etwas andere Form, bei der wir L und

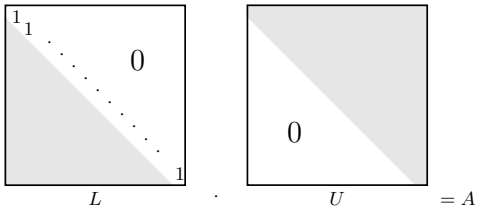


Abb. 14.4: LU-Zerlegung

U als *echte* untere bzw. obere Dreiecksmatrix auffassen und zusätzlich die Diagonalmatrix D und die Einheitsmatrix I verwenden:

$$A = (L \uplus I) \cdot (D \uplus U) \tag{14.3}$$

Danach kann man jedes gegebene Gleichungssystem $A \cdot \mathbf{x}_i = \mathbf{b}_i$ in zwei Schritten lösen, nämlich

$$(L \uplus I) \cdot \mathbf{y}_i = \mathbf{b}_i \qquad \text{und dann} \quad (D \uplus U) \cdot \mathbf{x}_i = \mathbf{y}_i \tag{14.4}$$

Im Folgenden diskutieren wir zuerst ganz kurz die Vorteile von Dreiecksmatrizen. Danach wenden wir uns dem eigentlichen Problem zu, nämlich der Programmierung der LU-Zerlegung.

14.4.1 Lösung von Dreieckssystemen

Weshalb sind Dreiecksmatrizen so günstig? Das macht man sich ganz schnell an einem Beispiel klar. Man betrachte das System

$$\begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ -2 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \\ 5 \end{pmatrix}$$

Hier beginnt man in der ersten Zeile und erhält der Reihe nach die Rechnungen

$$\begin{array}{rcl} 1 \cdot y_1 = 2 & \rightsquigarrow & y_1 = 2 \\ 3 \cdot y_1 + 1 \cdot y_2 = 6 & \rightsquigarrow & y_2 = 6 - 3 \cdot 2 = 0 \\ -2 \cdot y_1 + 2 \cdot y_2 + 1 \cdot y_3 = 5 & \rightsquigarrow & y_3 = 5 - (-2) \cdot 2 - 2 \cdot 0 = 9 \end{array}$$

Die allgemeine Situation ist in Abbildung 14.5 illustriert. Damit ergibt sich folgende Rechnung:

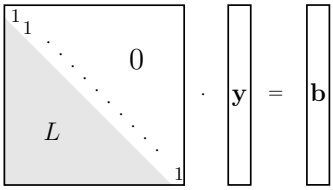


Abb. 14.5: Lösung eines (unteren) Dreiecksystems

$$(L \uplus I) \cdot \mathbf{y} = \mathbf{b} \tag{14.5}$$

Das lässt sich umformen in[5]

$$I \cdot \mathbf{y} = \mathbf{b} - L \cdot \mathbf{y} \tag{14.6}$$

Daraus folgt sofort

$$\mathbf{y} = \mathbf{b} - L \cdot \mathbf{y} \tag{14.7}$$

Diese Pseudo-Rekursion lässt sich aufgrund unserer Verabredungen über den Map-, Zip- und Reduce-Operator für Vektoren und Matrizen tatsächlich als Programm ausführen. Der entsprechende Code ist in Programm 14.9 angegeben (und wird weiter unten erläutert).

Völlig analog lässt sich das obere Dreieckssystem lösen. Aus

$$(D \uplus U) \cdot \mathbf{x} = \mathbf{y} \tag{14.8}$$

erhält man sofort

$$\mathbf{x} = D^{-1} \cdot (\mathbf{y} - U \cdot \mathbf{x}) \tag{14.9}$$

Auch diese Gleichung ist in Programm 14.9 implementiert.

Programm 14.9 Lösen eines (unteren) Dreieckssystems $L \cdot \mathbf{y} = \mathbf{b}$

```
FUN solveLower: Lower[n] × Vector[n] → Vector[n]
DEF solveLower(L, b) = y WHERE
                       y = b ⧑ (L · y)
FUN solveUpper: Diag[n] × Upper[n] × Vector[n] → Vector[n]
DEF solveUpper(D, U, y) = x WHERE
                          x = D⁻¹ · (y ⧑ (U · x))
```

[5] Der „⊎“-Operator erfüllt bzgl. „·“ das gleiche Distributivgesetz wie der „+“-Operator bei mathematischen Matrizen, weil er letztlich das Auffüllen mit „0“ repräsentiert.

Dieses Programm verwendet nur die Operatoren, die wir in Abschnitt 14.3 eingeführt haben. Die Effizienz hängt dabei ganz davon ab, wie gut der Compiler mit den speziellen Matrizen und mit Rekursionen der Bauart $y = b \overline{\bigvee} (L \cdot y)$ umgehen kann. Um das zu sehen, betrachten wir die Langform dieser Definition (die der Compiler intern daraus macht):

$$\begin{aligned} & y = b \overline{\bigvee} (L \cdot y) \\ \rightsquigarrow\ & y = \lambda\, i \bullet b(i) - (L.row\ i) \cdot y \end{aligned}$$

Weil L eine untere Dreiecksmatrix ist, hat die i-te Zeile den Definitionsbereich

$$Dom(L.row\ i) = (1..i-1)$$

Aufgrund der Definition des Zip-Operators wird bei der Bildung des Skalarprodukts y auf den gleichen Definitionsbereich eingegrenzt. Damit entsteht intern der Ausdruck

$$\rightsquigarrow y = \lambda\, i \bullet b(i) - (L.row\ i) \cdot (y \mid 1..i-1)$$

Weil $y(i)$ nur von Werten $y(j)$ abhängt, für die $j < i$ gilt, kann der Compiler erkennen, dass hier die aufsteigende Ordnung genommen werden muss. (Ansonsten müsste man den Overhead des Memoization-Mechanismus in Kauf nehmen.)

$$\rightsquigarrow y = \lambda\, i{:}\, (1 \nearrow n) \bullet b(i) - (L.row\ i) \cdot (y \mid 1..i-1)$$

14.4.2 *LU*-Zerlegung (Doolittle-Variante)

Bleibt also „nur“ noch das Problem, die Matrizen L und U zu finden. Wir wählen hier eine Variante der Gauß-Elimination, die auf Doolittle zurückgeht [121]. Die Grundidee ist in den Abbildungen 14.6 und 14.7 mit einem Schnappschuss der Berechnung skizziert. Bei diesem Schnappschuss gehen wir davon aus, dass die ersten $k-1$ Spalten von L und die ersten $k-1$ Zeilen von $D \uplus U$ schon berechnet sind.

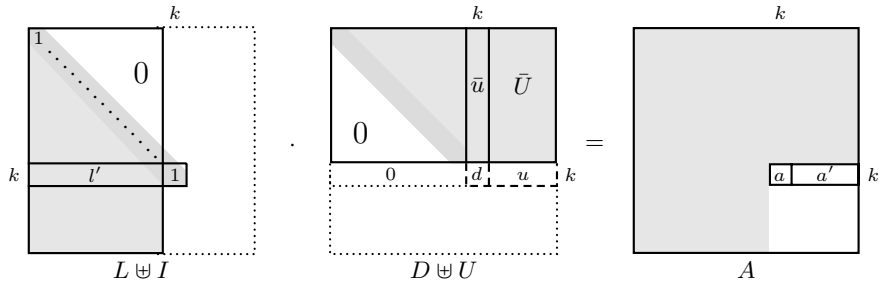


Abb. 14.6: LU-Zerlegung: Berechnung von u

Gemäß Abbildung 14.6 können wir die k-te Zeile von $D \uplus U$ nach folgenden Formeln bestimmen:

$$\begin{aligned} d &= a - l' \cdot \bar{u} \\ u &= a' - l' \cdot \bar{U} \end{aligned} \tag{14.10}$$

Übrigens sieht man hier sofort, dass die erste Zeile von $D \uplus U$ identisch ist mit der ersten Zeile von A, weil hier l' der leere Vektor ist.

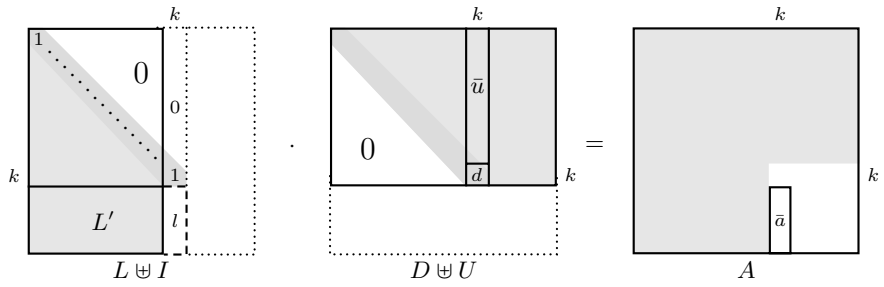


Abb. 14.7: LU-Zerlegung: Berechnung von l

Gemäß Abbildung 14.7 erhalten wir eine ähnliche Rechnung für die k-te Spalte von L:

$$l = \frac{1}{d}(\bar{a} - L' \cdot \bar{u}) \tag{14.11}$$

Diese Berechnungen lassen sich (optimistisch) direkt in das Programm 14.10 umsetzen.

Programm 14.10 Die LU-Zerlegung nach Gauß/Doolittle

```
FUN factor[n: Int]: Matrix[n, n] → Lower[n] × Diag[n] × Upper[n]
DEF factor(A) = (L, D, U)
    WHERE
    D          = λk • A(k, k) − L.row(k) · U.col(k)
    (rows U)   = λk • A.row(k) − L.row(k) · U.cols(k + 1..n)
    (cols L)   = λk • 1/D(k) · (A.col(k) − L.rows(k + 1..n) · U.col(k)
```

Damit diese naive Umsetzung funktioniert, brauchen wir die Konzepte aus den vorangehenden Abschnitten, die uns erlauben, Matrizen in Spalten- bzw. Zeilensicht zu definieren und mehrere Matrizen simultan (verschränkt rekursiv) zu definieren.

Ohne diese Konzepte würden die obigen Definitionen zu komplexer Memoization führen: Während des schrittweisen Aufbaus von D würden die entsprechenden Teile der Matrizen L und U peu à peu gesetzt werden. Am Ende der Besetzung von D wären auch L und U fertig – was der Compiler allerdings nicht wüsste.

Wenn man dem Compiler diese Fähigkeiten nicht zutraut, kann man den Prozess der simultanen Setzung der drei Matrizen auch „zu Fuß“ programmieren. Das sieht dann entsprechend hässlicher aus:

```
FUN iterate[n: Int]: Matrix[n, n] → Int → Triple → Triple
    WHERE Triple = Lower[n] × Diag[n] × Upper[n]
DEF iterate[n]A k (L, D, U) =
    IF k > n THEN (L, D, U)
    ELSE
    iterate[n]A k + 1 (L', D', U')
    WHERE
    D' = D k ⇐ A(k, k) − L.row(k) · U.col(k)
    U' = (rows U)k ⇐ A.row(k) − L.row(k) · (cols U) | (k + 1..n)
    L' = (cols L)k ⇐ 1/D(k) · (A.col(k) − (rows L) | (k + 1..n) · U.col(k)
```

14.4.3 Spezialfall: Gauß-Elimination bei Tridiagonalmatrizen

In einigen Anwendungen trifft man auf den Spezialfall von Gleichungssystemen mit *Tridiagonalmatrizen*. Ein Beispiel werden wir in Abschnitt 14.6 bei der Spline-Interpolation kennen lernen.

In Gleichung 14.10 sind l', $\bar{u}$ und $\bar{U}$ jeweils nur einelementig; Entsprechendes gilt für Gleichung 14.11. Damit vereinfacht sich das Programm 14.10 wesentlich. (Wir überlassen es dem interessierten Leser, die Details auszuprogrammieren.)

14.5 Beispiel: Interpolation

Naturwissenschaftler und Ingenieure sind häufig mit einem unangenehmen Problem konfrontiert: Man kennt nur ein paar Stichproben, also Messwerte $(x_0, y_0), \ldots (x_n, y_n)$, aber nicht die Funktion f, zu der diese Stichproben gehören. Trotzdem muss man den Funktionswert $\bar{y} = f(\bar{x})$ an einer gegebenen Stelle $\bar{x}$ ermitteln. Und diese Stelle $\bar{x}$ ist im Allgemeinen nicht unter den Stichproben enthalten. Diese Aufgabe der so genannten ***Interpolation*** ist in Abbildung 14.8 veranschaulicht: Die Messwerte $(x_0, y_0), \ldots, (x_n, y_n)$ werden als Stützstellen bezeichnet.

Wir gehen davon aus, dass der funktionale Zusammenhang „gutartig“ ist, d. h. durch eine möglichst „glatte“ Funktionskurve adäquat wiedergegeben wird. Da wir die Funktion f selbst nicht kennen, ersetzen wir sie durch eine andere Funktion p, die wir tatsächlich konstruieren können. Unter der Hypothese, dass f hinreichend „glatt“ ist, können wir p so gestalten, dass es sehr nahe an f liegt. Und dann berechnen wir die Approximation $\bar{y} = p(\bar{x}) \approx f(\bar{x})$.

Häufig nimmt man als Näherung p an die gesuchte Funktion f ein geeignetes Polynom. Ein ***Polynom*** vom Grad n ist ein Ausdruck der Form

$$p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_n \cdot x^n \tag{14.12}$$

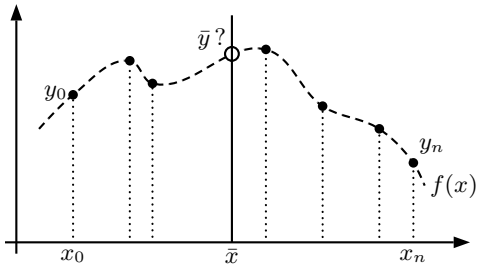


Abb. 14.8: Das Interpolationsproblem

mit gewissen Koeffizienten a_i. Das für unsere Zwecke grundlegende Theorem besagt, dass ein Polynom n-ten Grades durch $(n+1)$ Stützstellen eindeutig bestimmt ist. Bleibt also „nur" das Problem, das Polynom p zu berechnen. In anderen Worten: Wir müssen die Koeffizienten a_i bestimmen.

Die Lösungsidee

Newton hat ein cleveres Rechenverfahren für das Problem der Interpolation entwickelt, das unter dem Namen *dividierte Differenzen* in die Literatur eingegangen ist. Wir wollen hier nicht auf die mathematischen Details dieses Verfahrens eingehen (man findet sie z. B. in [134]), sondern nur diejenigen Aspekte zitieren, die für die Programmierung relevant sind. Man kann zeigen, dass das Polynom p auf folgende Weise erhalten werden kann.

$$\begin{aligned} p(x) = \; & a_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) + \\ & \ldots + \\ & a_2(x-x_0)(x-x_1) + \\ & a_1(x-x_0) + \\ & a_0 \end{aligned} \tag{14.13}$$

Bleibt das Problem, die Koeffizienten a_j auszurechnen. Die Idee von Newton organisiert diese Berechnung besonders geschickt und schnell, indem Koeffizienten $a_{i,j}$ nach folgender Rekurrenz bestimmt werden und $a_j = a_{0,j}$ gesetzt wird:

$$\begin{aligned} a_{i,i} &= y_i \\ a_{i,j} &= \tfrac{a_{i+1,j}-a_{i,j-1}}{x_j-x_i} \end{aligned} \tag{14.14}$$

Die Koeffizienten $a_{i,j}$ werden traditionell in der Form $f[x_i,\ldots,x_j]$ geschrieben und als *Newtonsche dividierte Differenzen* bezeichnet. Die Rekurrenzbeziehungen (14.14) führen zu den Abhängigkeiten, die in Abbildung 14.9 gezeigt sind. Man erkennt, dass die Koeffizienten $a_{i,j}$ als Elemente einer oberen Dreiecksmatrix gespeichert werden können. Die Diagonalelemente sind die Werte y_i und die erste Zeile enthält die gesuchten Koeffizienten des Polynoms (14.13).

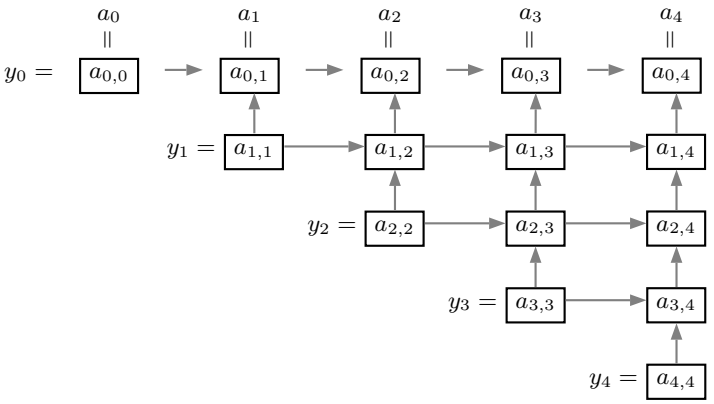


Abb. 14.9: Berechnungsschema der dividierten Differenzen

Das Programm

Das Programm 14.11 ist eine nahezu triviale Umsetzung der Strategie aus Abbildung 14.9 mit den Gleichungen (14.14).

Programm 14.11 Interpolation mit Newtonschen dividierten Differenzen

```
FUN interpolate[n: Nat]: Vector[0..n] × Vector[0..n] → (Number → Number)
DEF interpolate(x, y) = polynom
    WHERE
    A: (Diag[0..n] ⊎ Upper[1..n]) =
      λ i, j • IF i = j THEN y(i)
               IF i < j THEN (A.below(i, j) − A.left(i, j)) / (x(j) − x(i)) FI
    a = A.row(0)
    polynom = λ x • «Hornerschema mit Koeffizientenvektor a»
```

Die Funktion hat als Argumente die beiden Vektoren der x- und y-Koordinaten der Stützstellen und liefert als Resultat ein Polynom, das heißt eine Funktion *Number* → *Number*. (Die Polynomauswertung wird aus Effizienzgründen meistens nach dem so genannten Hornerschema vorgenommen, was wir hier nicht ausprogrammieren.) Ansonsten besteht das Programm nur aus der Angabe der Rekurrenzbeziehung 14.14; den Rest überlassen wir dem Compiler.

Damit kann man die Interpolation an einer Stelle $\bar{x}$ ganz einfach folgendermaßen schreiben:

$p(\bar{x})$ WHERE $p = interpolate(x, y)$

oder kurz

$interpolate(x, y)(\bar{x})$

Für die Berechnung der Matrix gibt es aufgrund der Abhängigkeiten aus Abbildung 14.9 drei Möglichkeiten; diese kann der Compiler anhand der Verwendung von *below* und *left* im Rumpf von Programm 14.11 erkennen:

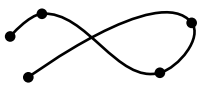
- Man kann Diagonale für Diagonale von unten nach oben berechnen.
- Man kann zeilenweise von unten nach oben und innerhalb jeder Zeile von links nach rechts arbeiten.
- Man kann spaltenweise von links nach rechts und innerhalb jeder Spalte von unten nach oben arbeiten.

Anmerkung 1: ***Vorsicht!*** *Die Werte $\bar{x}$, an denen man interpoliert, müssen* innerhalb *der Stützstellen $x_0, \ldots, x_n$ liegen. An den Rändern und vor allem außerhalb beginnt das Polynom im Allgemeinen stark zu oszillieren, so dass erratische Werte entstehen.*

Anmerkung 2: Im Programm 14.11 haben wir eine Matrix A benutzt. In Büchern zur Numerischen Mathematik findet man die Programme aber im Allgemeinen in einer Form, die mit einem eindimensionalen Array auskommt. Denn die Abhängigkeiten der Matrixfelder sind so, dass man immer alle tatsächlich noch benötigten Werte in einem Array halten kann. Angesichts der heutigen Speichergrößen spielen die paar Zellen aber keine Rolle mehr. Deshalb wäre es fast schon ein Kunstfehler, für diese Minioptimierung das Risiko eines falschen Programms einzugehen.

14.6 Beispiel: Spline-Interpolation

Moderne Graphik-Systeme verwenden zum Zeichnen von Kurven oft so genannte (parametrisierte) *Splines*. Das sind spezielle Polynome, die eine Menge gegebener Stützstellen besonders „glatt“ interpolieren. Wir beschränken uns im Folgenden auf das Grundproblem der normalen Spline-Interpolation, und zwar für den praktisch wichtigsten Fall der *kubischen Splines*. Wie üblich konzentrieren wir uns hier auf die Programmieraspekte; für die mathematischen Details verweisen wir auf Spezialliteratur der Numerik, z. B. [121, 73].



Anmerkung: Wir wollen wenigstens kurz erwähnen, wie die Grundform der Spline-Interpolation zum Zeichnen von Kurven der obigen Art verwendet werden kann. Man konstruiert aus der Punktemenge $(x_1, y_1), \ldots, (x_n, y_n)$ einen parametrischen Spline, indem man die Koordinaten als Funktionen einer Variablen t ansieht, also $x(t)$ und $y(t)$. Damit hat man die Stützstellen $x(t_1), \ldots, x(t_n)$ und $y(t_1), \ldots, y(t_n)$, für die man die Spline-Interpolation vornehmen kann. Für die Wahl geeigneter Werte t_i bieten sich an $t_0 = 0$ und $t_{i+1} = t_i + \delta_i$ mit $\delta_i = \ll$Abstand von (x_i, y_i) zu $(x_{i+1}, y_{i+1})\gg$.

Die Aufgabe der ***Spline-Interpolation*** ist folgendermaßen definiert: Gegeben ist eine Menge von $n + 1$ Stützstellen $(x_1, y_1), \ldots (x_{n+1}, y_{n+1})$. Im Gegensatz zur Newton-Interpolation suchen wir diesmal aber nicht ein einziges Polynom $p(x)$ vom Grad n, das alle $n + 1$ Punkte erfasst, sondern eine *Menge von Polynomen $s_1(x), \ldots, s_n(x)$ dritten Grades*, die das Gesamtintervall

stückweise abdecken (s. Abbildung 14.10). Genauer: Jedes $s_i(x)$ ist ein Polynom dritten Grades auf dem Intervall $[x_i..x_{i+1}]$.

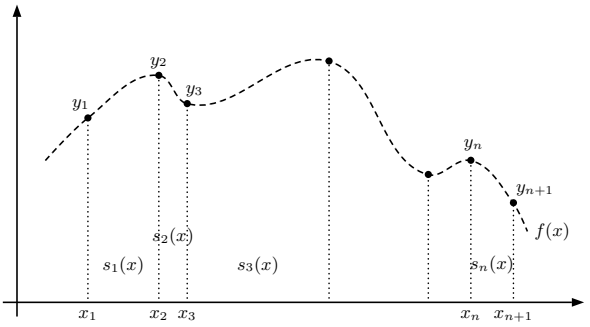


Abb. 14.10: Die Spline-Interpolation

Damit die Interpolation hinreichend „glatt“ ist, müssen benachbarte Polynome an ihrem Berührungspunkt sowohl im Wert als auch in der ersten und zweiten Ableitung übereinstimmen.

$$\begin{array}{ll}
s_i(x_i) = y_i, \quad s_i(x_{i+1}) = y_{i+1} & (i = 1, \ldots n) \\
s_i(x_{i+1}) = s_{i+1}(x_{i+1}) & (i = 1, \ldots, n-1) \\
s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}) & (i = 1, \ldots, n-1) \\
s_i''(x_{i+1}) = s_{i+1}''(x_{i+1}) & (i = 1, \ldots, n-1) \\
s_1''(x_1) = 0 & \text{(oder eine ähnliche Bedingung)} \\
s_n''(x_{n+1}) = 0 & \text{(oder eine ähnliche Bedingung)}
\end{array} \tag{14.15}$$

Die letzten beiden Bedingungen sind nötig, weil es sonst zu wenige Gleichungen gäbe, um die Lösung eindeutig zu machen. Für diese Zusatzbedingungen gibt es verschiedene Varianten; wir wählen hier diejenigen, die auf so genannte *natürliche Splines* führen.

Da die $s_i(x)$ kubische Polynome auf den Intervallen $[x_i..x_{i+1}]$ sind, stellen wir sie in folgender Form dar (für $i = 1, \ldots, n$):

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Zur besseren Lesbarkeit schreiben wir dies in Form von Vektoren und Matrizen. Damit haben die Funktion $s_i(x)$ und ihre beiden Ableitungen folgende Darstellung (für $i = 1, \ldots, n$):

$$\begin{bmatrix} s_i(x) \\ s_i'(x) \\ s_i''(x) \end{bmatrix} = \begin{bmatrix} 1 & (x - x_i) & (x - x_i)^2 & (x - x_i)^3 \\ 0 & 1 & 2(x - x_i) & 3(x - x_i)^2 \\ 0 & 0 & 2 & 6(x - x_i) \end{bmatrix} \cdot \begin{bmatrix} a_i \\ b_i \\ c_i \\ d_i \end{bmatrix} \tag{14.16}$$

Die Auswertung der drei Funktionen am linken Rand x_i liefert folgende Gleichungen (für $i = 1, \ldots, n$).

$$\begin{bmatrix} s_i(x_i) \\ s_i'(x_i) \\ s_i''(x_i) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_i \\ b_i \\ c_i \\ d_i \end{bmatrix} = \begin{bmatrix} a_i \\ b_i \\ 2c_i \end{bmatrix} \tag{14.17}$$

Die Auswertung der drei Funktionen am rechten Rand x_{i+1} liefert – wegen 14.17 – folgende Gleichungen (für $i = 1, \ldots, n$), wobei wir der Kürze halber $h_i := (x_{i+1} - x_i)$ verwenden.

$$\begin{bmatrix} s_i(x_{i+1}) \\ s_i'(x_{i+1}) \\ s_i''(x_{i+1}) \end{bmatrix} = \begin{bmatrix} 1 & h_i & h_i^2 & h_i^3 \\ 0 & 1 & 2h_i & 3h_i^2 \\ 0 & 0 & 2 & 6h_i \end{bmatrix} \cdot \begin{bmatrix} a_i \\ b_i \\ c_i \\ d_i \end{bmatrix} = \begin{bmatrix} a_{i+1} \\ b_{i+1} \\ 2c_{i+1} \end{bmatrix} \tag{14.18}$$

Damit dies auch für $i = n$ definiert ist, setzen wir $a_{n+1} := s_n(x_{n+1}) = y_{n+1}$, $b_{n+1} := s_n'(x_{n+1})$ und $c_{n+1} := \frac{1}{2} s_n''(x_{n+1})$.

Aus 14.15 und 14.17 folgt $a_i = y_i$. Und 14.18 lässt sich ausmultiplizieren und vereinfachen. Damit erhalten wir folgende Gleichungen (für $i = 1, \ldots, n$):

$$\begin{bmatrix} a_i \\ b_i + h_i c_i + h_i^2 d_i \\ 2c_i + 3h_i d_i \\ 3d_i \end{bmatrix} = \begin{bmatrix} y_i \\ \frac{1}{h_i}(a_{i+1} - a_i) \\ \frac{1}{h_i}(b_{i+1} - b_i) \\ \frac{1}{h_i}(c_{i+1} - c_i) \end{bmatrix} \tag{14.19}$$

Das sind $4n$ Gleichungen für die $4n + 2$ unbekannten Koeffizienten. (Denn wir haben zwei zusätzliche Koeffizienten b_{n+1} und c_{n+1} eingeführt. Wegen dieser Unterbestimmtheit brauchen wir die beiden Zusatzbedingungen aus 14.15.)

Dieses System könnte man jetzt mit Hilfe der Gauß-Elimination lösen. Aber aus Effizienzgründen rechnen wir noch ein bisschen symbolisch weiter, so dass unser Programm letztlich nur ein Gleichungssystem mit n Unbekannten lösen muss.

Wir setzen das a_i und das d_i in die zweite und dritte Zeile ein und lösen dann die zweite Zeile nach b_i auf:

$$\begin{bmatrix} a_i \\ b_i \\ c_{i+1} + c_i \\ d_i \end{bmatrix} = \begin{bmatrix} y_i \\ \frac{1}{h_i}(y_{i+1} - y_i) - \frac{h_i}{3}(c_{i+1} + 2c_i) \\ \frac{1}{h_i}(b_{i+1} - b_i) \\ \frac{1}{3h_i}(c_{i+1} - c_i) \end{bmatrix} \tag{14.20}$$

Damit sind die a_i direkt bekannt und die b_i und d_i hängen nur noch von den c_i ab. Also genügt es, die Koeffizienten c_i zu berechnen. Aus Symmetriegründen machen wir allerdings einen Indexshift; das heißt, wir betrachten für $i = 2, \ldots, n$ die Gleichungen

$$c_i + c_{i-1} \;=\; \frac{1}{h_{i-1}}(b_i - b_{i-1}) \tag{14.21}$$

In diese Gleichungen setzen wir jetzt unsere b_i ein. Danach ordnen wir sie so um, dass die Unbekannten c_{i-1}, c_i und c_{i+1} auf der linken Seite stehen:

$$h_{i-1}c_{i-1} + 2(h_{i-1}+h_i)c_i + h_i c_{i+1} = \frac{3}{h_i}(y_{i+1}-y_i) - \frac{3}{h_{i-1}}(y_i - y_{i-1}) \tag{14.22}$$

Da diese Gleichungen nur für die Indizes $i = 2, \ldots, n$ gelten, haben wir $n-1$ Gleichungen für die $n+1$ Unbekannten $c_1, \ldots, c_{n+1}$. Hier müssen wir die beiden Zusatzbedingungen aus 14.15 verwenden, die wegen 14.17 und unserer obigen Festlegung $c_{n+1} = \frac{1}{2}s_n''(x_{n+1})$ die Werte $c_1 = 0$ und $c_{n+1} = 0$ haben. Damit erhalten wir insgesamt das folgende Gleichungssystem:

$$\begin{bmatrix} 1 & 0 & & & \\ h_1 & 2(h_1+h_2) & h_2 & & \\ & \ddots & \ddots & \ddots & \\ & h_{n-1} & 2(h_{n-1}+h_n) & h_n & \\ & & 0 & 1 & \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{3}{h_2}(y_3-y_2) - \frac{3}{h_1}(y_2-y_1) \\ \vdots \\ \frac{3}{h_n}(y_{n+1}-y_n) - \frac{3}{h_{n-1}}(y_n - y_{n-1}) \\ 0 \end{bmatrix} \tag{14.23}$$

Dies ist ein Gleichungssystem für eine *Tridiagonalmatrix*, für das die LU-Zerlegung besonders einfach ist.

Das Programm 14.12 setzt die Gleichung 14.23 unmittelbar um. Zunächst wird die Matrix A als Tridiagonalmatrix aus den drei Diagonalvektoren aufgebaut. Die rechte Seite wird als Vektor z konstruiert.

Dann ergibt sich der Koeffizientenvektor als Ergebnis der Gauß-Elimination für $A \cdot c = z$.

Die Koeffizientenvektoren b und d ergeben sich gemäß Gleichung 14.20. Da $a_i = y_i$ gilt, können wir die Splines dann direkt als Array von Funktionen definieren.

Als Ergebnis müssen wir allerdings noch diese Splinepolynome mit den Stützpunkten $x_1, \ldots, x_{n+1}$ zusammenfassen, weil wir bei einer Interpolation an der Stelle $\bar{x}$ wissen müssen, welches der Splinepolynome zu verwenden ist.

Man beachte, dass wir im Programm 14.12 (zu Illustrationszwecken) zwei Stile nebeneinander verwenden. Die oberen Zeilen sind im kompakten Map-Filter-Reduce-Stil geschrieben (was manchmal Indexshifts erfordert), die unteren Zeilen sind punktuell mit expliziten Indizes formuliert.

Programm 14.12 Spline-Interpolation

```
TYPE Spline = (Real → Real)                                          -- Hilfstyp
FUN spline[n: Int]: Array[n+1]Point → Array[n]Spline × Array[1n+1]Real
DEF spline(points) = (splines, x)
    WHERE
    x = x ∗ points                                                   -- x-Koord.
    y = y ∗ points                                                   -- y-Koord.
    h = (x | 2..n+1)↓−1 ⩒ (x | 1..n)                                 -- ⇝ 1..n
    A: Tridiag[n+1] = (upper, diagonal, lower)
                      WHERE
                      upper = 0 ·: (h | 2..n)                        -- ⇝ 1..n
                      diagonal = 1 ·: (2 · (h↑+1 + (h | 2..n))) :· 1 -- ⇝ 1..n+1
                      lower = (h | 1..n−1) :· 0                      -- ⇝ 1..n
    z = 0 ·: (h′ ⩒ (y1 − y′) − h′′ ⩒ (y′ − y2)) :· 0                  -- ⇝ 1..n+1
        WHERE
        h′  = 3 · (1/h | 2..n)                                       -- ⇝ 2..n
        h′′ = 3 · (1/h | 1..n−1)↑+1                                  -- ⇝ 2..n
        y1  = (y | 3..n+1)↓−1                                        -- ⇝ 2..n
        y′  = (y | 2..n)                                             -- ⇝ 2..n
        y2  = (y | 1..n−1)↑+1                                        -- ⇝ 2..n
    c = gauss(A, z)
    a = y
    b = λ i • 1/h(i) · (y(i+1) − y(i)) − h(i)/3 · (c(i+1) + 2 · c(i))
    d = λ i • 1/(3·h(i)) · (c(i+1) − c(i))
    splines = λ i •
      (λ x̄ • a(i) + b(i) · (x̄ − x(i)) + c(i) · (x̄ − x(i))^2 + d(i) · (x̄ − x(i))^3)
```

14.7 Beispiel: Schnelle Fourier-Transformation (FFT)

Wir hatten in Abschnitt 14.5 bereits festgestellt, dass ein ***Polynom*** vom Grad $n-1$ auf zwei gleichwertige Arten eindeutig beschrieben werden kann:

- als Ausdruck der Form $p(x) = a_{n-1} \cdot x^{n-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$ mit gewissen Koeffizienten a_i;
- durch n Stützstellen $(x_1, y_1), \ldots (x_n, y_n)$.

Oft muss man aber mit den Polynomen weiter arbeiten. Typische Aufgaben sind dabei die Bildung der Summe, der Differenz, des Produkts oder des Quotienten zweier Polynome p und q. Dabei zeigt sich, dass die beiden Formen der Polynomdarstellung nicht für alle Zwecke gleichermaßen geeignet sind:

Darstellung	Auswertung	Addition	Multiplikation	Konversion
Koeffizienten				?
Stützstellen				?

Die *Auswertung* von Polynomen ist in der Koeffizientendarstellung schnell und einfach (und erfolgt üblicherweise nach dem so genannten Horner-Schema), während man bei der Stützstellenform ein aufwendiges Verfahren wie z. B. Newtons dividierte Differenzen braucht (vgl. Abschnitt 14.5). Die *Addition* (und Subtraktion) ist in beiden Repräsentationen einfach: In der Koeffizientenform addiert man einfach die entsprechenden Koeffizienten, in der Stützstellenform addiert man die entsprechenden y-Werte – vorausgesetzt, beide Polynome haben ihre Stützstellen an den gleichen Postionen $x_1, \ldots, x_n$. Die *Multiplikation* (und Division) ist in der Koeffizientenform teuer; sie erfordert $\mathcal{O}(N^2)$ Operationen. In der Stützstellenform muss man dagegen wieder nur die entsprechenden y-Werte miteinander multiplizieren, was mit linearem Aufwand $\mathcal{O}(N)$ möglich ist.

Da keine der beiden Darstellungen für alle Aufgaben geeignet ist, wäre es gut, wenn man einigermaßen schnell zwischen beiden Formen hin und her wechseln könnte. Eine Möglichkeit dazu liefert die so genannte schnelle ***Fourier-Transformation*** (kurz: ***FFT***) von Cooley und Tukey. Allerdings geht das nur, wenn man die Freiheit hat, sehr spezielle Stützstellen zu wählen.

Die Mathematik: n-te Einheitswurzeln

Das wesentliche Konzept im Zusammenhang mit FFT sind die so genannten *n-ten Einheitswurzeln*. Das sind komplexe Zahlen z mit der Eigenschaft $z^n = 1$. Es ist bekannt, dass es zu jedem $n \in \mathbb{N}$ genau n solche Einheitswurzeln gibt (die alle verschieden sind). Eine davon ist die prinzipielle Einheitswurzel, die wir als $root(n)$ bezeichnen. Sie ist definiert durch

$$r \stackrel{def}{=} root(n) \stackrel{def}{=} e^{\frac{i2\pi}{n}} = \cos(\frac{2\pi}{n}) + i\sin(\frac{2\pi}{n}) \tag{14.24}$$

Ausgehend von r erhalten wir alle n Einheitswurzeln durch die Potenzen $r^0, r, r^2, \ldots, r^{n-1}$. Für *gerades* n folgt aus der Definition 14.24 sofort:

$$\begin{array}{lcll} r^m & = & -1 & \text{für } n = 2m \\ r^{m+k} & = & -r^k & \text{für } n = 2m, 0 \leq k < m \\ r^2 & = & s & \text{für } n = 2m, s = root(m) \end{array} \tag{14.25}$$

Von den Koeffizienten zu den Stützstellen

Generell gilt: Um ein Polynom $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ vom Grad $n-1$ an den n Stützpunkten $x_0, \ldots, x_{n-1}$ auszurechnen, muss man im Allgemeinen n Auswertungen vornehmen:

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} p(x_0) \\ \vdots \\ p(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ & \vdots & & \vdots & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} \quad (14.26)$$

In Kurzform:

$$\mathbf{y} = X \cdot \mathbf{a} \quad (14.27)$$

Die Matrix X der Potenzen der Stützpunkte $x_0, \dots, x_{n-1}$ heißt *Vandermonde-Matrix*.

Entscheidend bei der schnellen Fourier-Transformation ist, dass man *die n-ten Einheitswurzeln als Stützstellen für das Polynom* wählt. Zur besseren Lesbarkeit verwenden wir die Exponenten als Indizes, d. h., wir setzen $r_j \stackrel{def}{=} r^j$. Die generelle Gleichung 14.26 erhält damit die besondere Form

$$\begin{bmatrix} b_0 \\ \vdots \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} p(r_0) \\ \vdots \\ p(r_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & r_0 & r_0^2 & \dots & r_0^{n-1} \\ & \vdots & & \vdots & \\ 1 & r_{n-1} & r_{n-1}^2 & \dots & r_{n-1}^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} \quad (14.28)$$

In Kurzform:

$$\mathbf{b} = F^{(n)} \cdot \mathbf{a} \quad (14.29)$$

Die Vandermonde-Matrix $F^{(n)}$ der Potenzen der n-ten Einheitswurzeln r_0, $\dots$, r_{n-1} heißt *Fouriermatrix*.

Eine effiziente Berechnungsstrategie

Wenn man Programme effizienter machen will, sollte man prüfen, ob man Mehrfachberechnungen einsparen kann. Die speziellen Eigenschaften der Einheitswurzeln, die in den Gleichungen 14.25 angegeben sind, bewirken solche Mehrfachberechnungen. Dies ist in Abbildung 14.11 (für den Fall $n = 8$) skizziert.

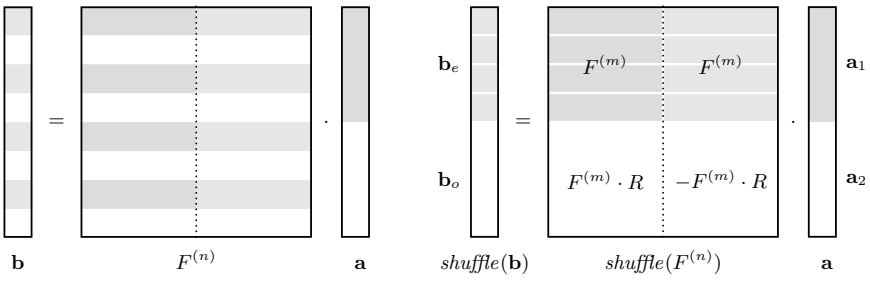


Abb. 14.11: Berechnungsmuster der FFT

Zeilen mit geraden Indizes in $\mathbf{b}$ und $F^{(n)}$ sind grau unterlegt. Die Operation *shuffle* sortiert sie so um, dass diese Zeilen an die Stellen $0, \ldots, m-1$ (für $n = 2m$) wandern (also gerade nach vorne, ungerade nach hinten).

Außerdem splitten wir die Matrix in die linke Hälfte mit den Spalten $0, \ldots, m-1$ und die rechte Hälfte mit den Spalten $m, \ldots, n-1$; der Vektor $\mathbf{a}$ wird entsprechend halbiert.

Damit entstehen insgesamt vier Teilmatrizen $F_{1,1}$, $F_{1,2}$, $F_{2,1}$ und $F_{2,2}$. Diese haben aufgrund der Gleichungen 14.25 folgende Eigenschaften (mit $n = 2m$):

- $F_{1,1} = F^{(m)}$
  Das ergibt sich aus der Definition $r_j = r^j$ und der Eigenschaft $r^2 = s$. Denn die linke Hälfte einer geraden Zeile hat die Form $\begin{bmatrix} r_{2k}^0 & \ldots & r_{2k}^j & \ldots & r_{2k}^{m-1} \end{bmatrix} = \begin{bmatrix} r^{2k\cdot 0} & \ldots & r^{2kj} & \ldots & r^{2k(m-1)} \end{bmatrix} = \begin{bmatrix} s_k^0 & \ldots & s_k^j & \ldots & s_k^{m-1} \end{bmatrix}$.
- $F_{1,2} = F^{(m)}$
  Dies folgt analog zur obigen Rechnung mit der zusätzlichen Eigenschaft $r^{2km} = r^{nk} = 1^k = 1$. Damit ist dann $r_{2k}^{m+j} = s_k^j$.
- $F_{2,1} = F^{(m)} \cdot R$
  Diese Rechnung ist ein kleines bisschen aufwendiger. Die linke Hälfte einer ungeraden Zeile hat die Form $\begin{bmatrix} r_{2k+1}^0 & \ldots & r_{2k+1}^j & \ldots & r_{2k+1}^{m-1} \end{bmatrix}$. Die obigen Eigenschaften ergeben $r_{2k+1}^j = r^{2kj+j} = s_k^j \cdot r^j$. Deshalb müssen wir die Matrix $F^{(n)}$ noch von rechts mit der Diagonalmatrix R multiplizieren, die die Potenzen $r^0, \ldots, r^{m-1}$ enthält.

$$R \stackrel{def}{=} \begin{bmatrix} r^0 & & & & \\ & r^1 & & & \\ & & \ddots & & \\ & & & & r^{m-1} \end{bmatrix}$$

- $F_{2,2} = -F^{(m)} \cdot R$
  Das folgt analog zur obigen Rechnung. Bei den Elementen r_{2k+1}^{m+j} kommt aber noch der Faktor $r^m = -1$ hinzu.

Insgesamt ergibt sich aus Abbildung 14.11 folgende Gleichung:

$$\mathit{shuffle}(\mathbf{b}) = \begin{bmatrix} \mathbf{b}_e \\ \mathbf{b}_o \end{bmatrix} = \begin{bmatrix} F^{(m)} \cdot (\mathbf{a}_1 + \mathbf{a}_2) \\ F^{(m)} \cdot R \cdot (\mathbf{a}_1 - \mathbf{a}_2) \end{bmatrix} \tag{14.30}$$

Damit haben wir die Berechnung der FFT vom Grad n rekursiv zurückgeführt auf die Berechnung einer FFT vom Grad $m = \frac{n}{2}$. Das ist bekanntlich der wesentliche Schritt, um zu einem Verfahren der Ordnung $\mathcal{O}(\log n)$ zu kommen.

Das Programm

Nach der mathematischen Vorarbeit ist die Programmierung jetzt sehr einfach. Um den Wert $\mathit{fft}(a)$ zu berechnen, bilden wir die zwei Teilvektoren $(a', a'') = ((a_1 + a_2),\ R \cdot (a_1 - a_2))$ und berechnen $\mathit{fft} * (a', a'')$. Danach muss noch die Inverse von *shuffle*, also *unshuffle*, ausgeführt werden.

Man beachte: Aufgrund unserer Konvention, dass Vektoren und Matrizen mit $1..n$ nummeriert werden, ergeben sich hier gegenüber den obigen Formeln (die mit $0..n-1$ nummerieren) leichte Umindizierungen; insbesondere vertauschen die Namen b_e und b_o leider ihre mnemotechnische Bedeutung. (Aber Abbildung 14.11 gibt die richtige Intuition.)

Programm 14.13 Berechnung der FFT

```
STRUCTURE FastFourierTransformation = {
  FUN fft[n: Nat]: Vector[n] → Vector[n]
  DEF fft(a) =  unshuffle(b_e, b_o)
                WHERE
                b_e = fft(upper a + lower a)
                b_o = fft(R · (upper a − lower a))
                m = n/2
                upper a = (a | 1..m)
                lower a = (a | m+1..n)↓m
                R: DiaMatrix[m] = λk • r^(k−1) WHERE r = e^(i2π/n)
  FUN unshuffle: Vector[m] × Vector[m] → Vektor[n]
  DEF unshuffle(u, v) = λi • IF odd(i) THEN u((i+1)/2)
                               ELSE v(i/2)  FI
}
```

Für die Berechnung von R kann man auch eine effizientere Form wählen, die jedes Element mit einer Multiplikation aus dem vorausgehenden ableitet (im Stil vergleichbar mit den Programmen über lazy Listen):

$$R = 1 \cdot\!: ((r\cdot) * (R \mid 1..m-1))^{\uparrow+1}$$

Man sieht übrigens auch, dass hier Vektoren gar nicht unbedingt nötig wären; man könnte das Programm auch über Listen implementieren.

15

Map: Wenn Funktionen zu Daten werden

Das Gedächtnis ist nach Gegenständen verteilt, und in niemand ist es für alle gleich gut.

W. von Humboldt

Funktionen sind *das* zentrale Konzept funktionaler Sprachen. Deshalb ist es eigentlich überraschend, dass sie in fast allen Sprachen sehr engherzig betrachtet werden. Funktionen sind letztlich λ-Ausdrücke (für Theorie-interessierte Programmierer) bzw. Routinen in Maschinencode (für compilertechnisch orientierte Programmierer). Diese grundlegende Sicht wird dann noch mit ein bisschen syntaktischem Zuckerguss (engl.: *syntactic sugar*) überzogen wie z. B. Infix- oder Mixfix-Notationen, Funktionen höherer Ordnung etc.

Davon losgelöst ist ein anderes Konzept, das unter Namen wie *Tables* oder *Maps* firmiert und das dem Bereich der Datenstrukturen zugeordnet wird. Als Implementierungen hat man üblicherweise Suchbäume oder Hash-Tabellen, in einfachen Fällen auch nur Listen von Paaren. Typische Anwendungen sind z. B. die Zuordnung von Kundendaten zu Kundennummern in Geschäftsapplikationen oder die Zuordnung von Typinformationen zu Identifiern in Compilern.

Aber bei genauerem Hinsehen stellt man fest, dass auf der konzeptuellen Ebene zwischen den beiden Dingen kein Unterschied erkennbar ist. Die Differenzierung existiert „nur“ in der Implementierung. Deshalb liegt es nahe, beide Konzepte auf der Ebene der Programmierung zu verschmelzen und die Differenzierung dem Optimierungsteil des Compilers zu überlassen.

Das gleiche Phänomen hatten wir schon bei den Arrays kennen gelernt (vgl. Kapitel 14). Auch dort hatte sich eine klassische Datenstruktur der gängigen Programmiersprachen letztlich nur als Variation der Funktionen entpuppt. Deshalb ist es jetzt an der Zeit, dieses Phänomen grundsätzlicher anzugehen und die Gemeinsamkeiten und Unterschiede der verschiedenen Konzepte herauszuarbeiten.

15.1 Variationen über Funktionen

Wir haben bisher Funktionen als ein fundamentales mathematisches Phänomen akzeptiert, das als primitives Konzept in die Sprache eingebaut ist. Notiert haben wir Funktionen ausschließlich in der Form von λ-Ausdrücken bzw. dazu äquivalenten syntaktischen Spielarten. Aber in der Mathematik werden Funktionen eigentlich als spezielle (nämlich rechtseindeutige) Relationen eingeführt. Das wiederum ist gleichwertig zu Mengen von Paaren. Und dem entsprechen in der Informatik eher Datenstrukturen wie Tabellen oder Suchbäume. Mit anderen Worten: Der Funktionsbegriff ist durchaus schillernd und erlaubt unterschiedliche Sichtweisen.

Im Folgenden betrachten wir die verschiedenen Sichten auf Funktionen etwas detaillierter, was zu folgenden Unterscheidungen führt:

- *λ-Terme* sind die klassische „operationale“ Sicht von Funktionen in Programmiersprachen. Die Details des operationalen Verhaltens spiegeln sich in den verschiedenen Auswertungsstrategien und der dadurch induzierten Semantik wider (vgl. Abschnitt 1.3).
- *Tabellarische Auflistungen* sind eine Datenstruktur-orientierte Sicht auf Funktionen. Wir verwenden dafür auch die (schon in Kapitel 14 eingeführte) Metapher der „eingefrorenen“ Funktionen. Dabei muss man pragmatisch zwei Situationen unterscheiden:
  - *Dichte* Funktionen haben einen Definitionsbereich, der unmittelbar einem kompakten Intervall der natürlichen Zahlen entspricht. Dies führt auf die bekannten Arrays (vgl. Kapitel 14).
  - *Gestreute* Funktionen haben einen Definitionsbereich, der nur einen kleinen Teil einer im Allgemeinen riesigen Grundmenge ausmacht. Ein typisches Beispiel ergibt sich, wenn eine Versicherung zwölfstellige Schlüssel als Kundennummern verwendet, obwohl die Zahl der Kunden nicht einmal theoretisch die Billionengrenze erreichen kann. Dies führt auf Konzepte wie Hash-Tabellen oder Rot-Schwarz-Bäume [37, 129].

Diese Aspekte wollen wir jetzt auch in der Programmierung etwas deutlicher reflektieren. Deshalb führen wir entsprechende Typklassen ein:

$$\mathit{Fun} \supseteq \mathit{Map} \supseteq \mathit{Array} \quad -- \textit{ Hierarchie von Typklassen für Funktionen}$$

Diese Hierarchie spiegelt folgende Eigenschaften wider:

- Die Typklasse *Fun* umfasst alle Funktionen.
- Die Typklasse *Map* umfasst die „eingefrorenen“ Funktionen.
- Die Typklasse *Array* umfasst die „eingefrorenen“ Funktionen, die dicht sind, d. h. deren Definitionsbereiche Intervalle sind.

Insgesamt erhalten wir ein Package, wie es in Programm 15.1 skizziert ist. (Dabei sollte das Overloading des Namens *Functions* harmlos sein.) In diesem Package sind alle „funktionsartigen“ Strukturen zusammengefasst. Da

Programm 15.1 Das Package der funktionsartigen Strukturen

```
PACKAGE Functions = {
  SPECIFICATION Functions = {...}
  TYPECLASS Fun = Functions.Fun
  SPECIFICATION Mappings = {...}
  TYPECLASS Map = Mappings.Fun
  PROP Map ⊆ Fun
  SPECIFICATION Arrays = {...}
  TYPECLASS Array = Arrays.Array
  PROP Array ⊆ Map
  SPECIFICATION Intervals = {...}
  TYPECLASS Interval = Intervals.Interval
}
```

ihre Implementierungen aber im Compiler verborgen sind, geben wir hier nur die entsprechenden Spezifikationen an.

Die Spezifikation *Arrays* und die zugehörige Spezifikation *Intervals* haben wir schon in Kapitel 14 eingehend studiert. Deshalb konzentrieren wir uns jetzt auf die Spezifikationen *Mappings* und *Functions* und ihre zugehörigen Typklassen *Fun* und *Map*.

15.2 Die Typklasse der Funktionen

Grundsätzlich ist das Konzept der Funktionen in der Sprache und damit im Compiler vorgegeben. Im Programm 15.2 geben wir deshalb nur eine (partielle) Spezifikation der entsprechenden Typklasse und der wichtigsten Operatoren auf Funktionen an.[1] Die Liste in Programm 15.2 ließe sich noch weiter fortsetzen; aber wir verzichten auf die Angabe weiterer Operationen wie z. B. Currying oder Uncurrying, weil diese bereits in Kapitel 1 in der Struktur *HigherOrder* von Programm 1.1 definiert wurden.

Wir gehen davon aus, dass der Pfeil-Operator $\_ \to \_$ auf allen Ebenen der Typisierung vom Compiler vorgegeben ist. Um die Typklasse *Fun* leichter definieren zu können, führen wir einen speziellen Typoperator *Fun* als Instanz des Pfeil-Operators auf der Ebene der Typen ein. Aus Gründen der Lesbarkeit bleiben wir aber bei allen Applikationen bei der Pfeil-Notation.

Bei der Definition der Restriktion ist die IF-Abfrage eigentlich unnötig, weil diese IF-Abfrage als Teil der (dynamischen) Typisierung ohnehin generiert wird. Aber in dieser Form ist die Semantik klarer sichtbar. Bei der Typisierung der Restriktion müssen wir das Prinzip der abhängigen Typisierung auf die

[1] Aus Gründen der Lesbarkeit kennzeichnen wir die polymorphen Typvariablen wieder nur durch die Verwendung griechischer Buchstaben.

Programm 15.2 Die Typklasse der Funktionen

```
SPECIFICATION Functions = {
  TYPE Fun: Type → Type → Fun                                  -- Typkonstruktor
  DEF Fun α β = (α → β)
  TYPE Dom: Fun → Type                                         -- Domain
  TYPE Ran: Fun → Type                                         -- Range
  DEF Dom(α → β) = α
  DEF Ran(α → β) = β
  FUN _ _: (α → β) → α → β                                     -- Applikation
  FUN _ ∘ _: (β → γ) × (α → β) → (α → γ)                       -- Komposition
  FUN _ | _: (α → β) × (γ̂: Type • γ ⊆ α) ↦ (γ → β)             -- Restriktion
  FUN id: α → α                                                -- Identität
      ⋮      (s. Programm 1.1)
  DEF f(a) = GIVEN                                             -- Applikation
  DEF (g ∘ f)(a) = g(f(a))                                     -- Komposition
  DEF (f | γ)(a) = IF a:γ THEN f(a) FI                         -- Restriktion
  DEF id(a) = a                                                -- Identität
      ⋮      (s. Programm 1.1)
}
TYPECLASS Fun = Functions.Fun
```

Ebene der Klassen anheben, weil das zweite Argument γ von $f \mid \gamma$ ein Typ ist, der Subtyp des Definitionsbereichs $Dom(f)$ sein muss.

Anmerkung: Die Typklasse **Fun** *repräsentiert eine coalgebraische Sicht auf Funktionen: Wir haben eine Reihe von Operationen, die Funktionen verwenden („Beobachter"), aber es gibt keine Operationen zur Konstruktion von Funktionen. Diese sind im Compiler verborgen; das gilt insbesondere für die konkrete Implementierung von Funktionen über λ-Ausdrücke (und dazu äquivalente syntaktische Konstruktionen).*

Die Typklasse *Fun* umfasst alle Funktionen, ohne irgendwelche Restriktionen. Das heißt, sowohl Definitions- als auch Wertebereich sind beliebige Typen (also von der Art *Type*). Im Folgenden werden wir uns mit speziellen Teilklassen von Funktionen befassen, bei denen der Definitionsbereich gewissen Einschränkungen genügen muss.

15.3 Die Typklasse der Mappings

Neben der klassischen Repräsentation über λ-Ausdrücke (also über „Code") gibt es auch den konstruktiven Aufbau von Funktionen durch schrittweises Hinzufügen von Wertepaaren, beginnend mit der „leeren" Funktion.

Für diese konstruktiv aufgebauten (und damit endlichen) Funktionen hat sich der Begriff *Map* eingebürgert. Solche Maps sind nicht mehr über be-

liebigen Typen für den Definitionsbereich definierbar, sondern nur noch über Typen der Klasse *Eq*, weil für die Implementierung der Applikation ein Gleichheitstest gebraucht wird.

Programm 15.3 zeigt die wichtigsten Operatoren für Maps; Programm 15.4 enthält die wesentliche Eigenschaften dieser Operatoren. (Wegen der Länge mussten wir die Spezifikation auf zwei Seiten verteilen.)

Wegen der Eigenschaft $Map \subseteq Fun$ sind auf *Map* auch alle Operationen verfügbar, die auf *Fun* definiert wurden, also insbesondere die Applikation $m(a)$, die Komposition $m_1 \circ m_2$ und die Restriktion $m \mid \gamma$.

Programm 15.3 Die Typklasse *Map* und der Datentyp *Map*

```
SPECIFICATION Mappings = {
  TYPE Map: Eq → Type → Map                                   -- Typkonstruktor
  DEF Map α β = (α → β)
  PROP Map α β ⊆ Fun α β
  FUN □: Map α β                                              -- leere Map
  FUN _ ↦ _: α × β → Map α β                                  -- Singleton-Map
  FUN _ ⫧ _: Map α β × Map α β → Map α β                      -- Überschreiben
  FUN _ _ ⇐ _: Map α β × α × β → Map α β                      -- Überschreiben
  FUN _ ∪ _: Map α β × Map α β → Map α β  VAR β: Eq           -- Vereinigung
  FUN _ − _: Map α β × Set α → Map α β                        -- Löschen
  FUN _ − _: Map α β × α → Map α β                            -- Löschen
  FUN _ | _: Map α β × Set α → Map α β                        -- Restriktion
  FUN dom _: Map α β → Set α                                  -- Definitionsbereich
  FUN ran _: Map α β → Set β                                  -- Wertebereich
  FUN _ ∈ _: α × Map α β → Bool                               -- definiert?
  FUN _ = _: Map α β × Map α β → Bool  VAR β: Eq              -- gleich?
  FUN _ ≠ _: Map α β × Map α β → Bool  VAR β: Eq              -- ungleich?
  FUN _ ⊆ _: Map α β × Map α β → Bool  VAR β: Eq              -- Teilmap?
    ⋮    (Fortsetzung in Programm 15.4)
}
TYPECLASS Map = Mappings.Map
PROP Map ⊆ Fun
```

Man beachte, dass wir aus Gründen der Lesbarkeit in allen Operationen die explizite Kennzeichnung der Typvariablen α und β weglassen. Streng genommen müssten wir z. B. die Singleton-Operation folgendermaßen schreiben:

```
FUN _ ↦ _: α × β → Map α β   VAR α: Eq, β: Type
```

Da wir offen lassen wollen, wie Maps implementiert werden (Hash-Tabellen, Rot-Schwarz-Bäume etc.), charakterisieren wir (in Programm 15.4) die Operationen nur über Propertys und nicht über konkrete Definitionen.

Programm 15.4 *Mappings* (Fortsetzung)

```
SPECIFICATION Mappings = {
                ⋮      (Fortsetzung von Programm 15.3)
  PROP (a ↦ b)(a') = IF a = a' THEN b FI
  PROP (m1 ⫧ m2)(a) = IF a ∈ m2 THEN m2(a) ELSE m1(a) FI
  PROP (m a ⇐ b) = (m ⫧ (a ↦ b))
  PROP (m − s)(a) = IF a ∉ s THEN m(a) FI
  PROP (m − a) = m − {a}
  PROP (m | s) = (m − (dom(m) − s))

  PROP dom □ = ∅
  PROP ran □ = ∅
  PROP dom(a ↦ b) = {a}
  PROP ran(a ↦ b) = {b}
  PROP dom(m1 ⫧ m2) = dom(m1) ∪ dom(m2)
  PROP ran(m1 ⫧ m2) = ran(m1 − dom(m2)) ∪ ran(m2)
  PROP dom(m − s) = dom(m) − s

  PROP a ∈ m = a ∈ dom m

  PROP m1 = m2 ⟺ dom(m1) = dom(m2) ∧ ∀x ∈ dom m1 • m1(x) = m2(x)
  PROP m1 ⊆ m2 ⟺ dom(m1) ⊆ dom(m2) ∧ (m2 | dom(m1)) = m1

  PROP m1 ∪ m2 REQUIRES m1 | d = m2 | d WHERE d = dom(m1) ∩ dom(m2)
  PROP (m1 ∪ m2) = (m1 ⫧ m2)
  PROP (m1 ⫧ m2) = (m1 − dom(m2)) ∪ m2

  PROP (m1 ⫧ (m2 ⫧ m3)) = ((m1 ⫧ m2) ⫧ m3)      −− assoziativ
  PROP m1 ∪ (m2 ∪ m3) = (m1 ∪ m2) ∪ m3          −− assoziativ
  PROP m1 ∪ m2 = m2 ∪ m1                        −− kommutativ
}
```

Maps werden konstruktiv aufgebaut. Ausgehend von der leeren Map □ und der einelementigen Map $(a \mapsto b)$ baut man mit Hilfe des Kompositionsoperators $(m_1 \overleftarrow{+} m_2)$ immer größere Maps auf. Dabei ist $\overleftarrow{+}$ im Wesentlichen die Vereinigung der beiden Maps; auf dem Durchschnitt ihrer Definitionsbereiche dominiert allerdings m_2, weshalb der Name „Überschreiben“ gerechtfertigt ist. Für den häufigen Fall des Überschreibens eines einzelnen Wertes haben wir die spezielle Notation $m\ a \Leftarrow b$ eingeführt. Man beachte aber, dass beim Überschreiben – zumindest konzeptionell – eine komplette neue Map generiert wird. (Ob der Compiler das intern durch echtes Überschreiben oder durch Kopieren implementiert, hängt von der Single-Threadedness ab.)

Da *Map* eine Subklasse von *Fun* ist, gibt es auf Maps die Operatoren *Dom* und *Ran*, die den Argumenttyp α und den Wertetyp β liefern. Wenn wir aber die tatsächlichen Definitions- und Werte*mengen* brauchen, verwenden wir die

Operatoren *dom* und *ran*. (Weil diese nicht Typen, sondern Werte liefern, schreiben wir sie klein.)

Die Operatoren $(m - s)$ und $(m - a)$ erlauben es, Maps zu verkleinern, indem man eine Teilmenge s bzw. ein Element a aus dem Definitionsbereich entfernt. Die Restriktion $m \mid s$ einer Map m auf eine Menge s lässt nur diejenige Teilmap übrig, deren Definitionsbereich $s \cap dom(m)$ ist.

Weil die Operationen *dom* und *ran* unter Umständen sehr große Mengen berechnen, ist ein Test $x \in dom\ m$ im Allgemeinen sehr aufwendig. Da er aber häufig gebraucht wird, führen wir dafür eine effizientere Operation $x \in m$ ein, die den Test direkt *implementiert* (allerdings genau die Property $x \in m \Longleftrightarrow x \in dom\ m$ erfüllt).

In manchen Situationen muss man prüfen, ob zwei Maps gleich sind oder ob eine Map in einer anderen Map enthalten ist; dazu dienen die Operation $m_1 = m_2$ und $m_1 \subseteq m_2$.

Weil der Operator $m_1 \overleftarrow{+} m_2$ nur assoziativ ist, möchte man auch eine kommutative Variante haben. Diese ist aber nur unter gewissen Einschränkungen wohl definiert: Die Vereinigung $m_1 \cup m_2$ ist nur dann sinnvoll, wenn m_1 und m_2 auf ihrem Durchschnitt übereinstimmen.

Man beachte, dass die Operation $m_1 = m_2$ und die davon abhängigen Operationen $m_1 \neq m_2$, $m_1 \subseteq m_2$ und $m_1 \cup m_2$ nur definiert sind, wenn der Wertebereich der beiden Maps ein Typ der Klasse *Eq* ist.

Zur Erinnerung: Wir hatten in Kapitel 10 auch noch weitere Operationen auf Maps eingeführt, durch die Maps zu CPOs wurden.

15.3.1 Implementierungen von Maps

Im Unterschied zu Arrays wollen wir bei Maps nahezu beliebige Definitionsbereiche zulassen, was das Speichern komplexer macht. Völlig beliebig dürfen wir allerdings nicht werden, denn einige Einschränkungen sind – abhängig von der gewünschten Implementierung – doch nötig. Wenn *Map* α β als Liste von Paaren dargestellt wird, muss α: *Eq* gelten; wenn Suchbäume (z. B. Rot-Schwarz-Bäume) verwendet werden, muss α: *Ord* gelten; und bei der effizientesten Variante, der Darstellung mittels Hash-Tabellen, muss α: *Hash* gelten, mit einer geeigneten Typklasse *Hash*. (Die jeweiligen Implementierungstechniken findet man in der Standardliteratur zu Datentypen, z. B. [37, 129].)

15.3.2 Map-Filter-Reduce auf Maps

Wie bei allen Datenstrukturen sollte man auch bei Maps die elementaren Funktionen höherer Ordnung bereitstellen, mit denen eine kompakte Programmierung nach dem *Map-Filter-Reduce*-Paradigma möglich ist. Die entsprechenden Operationen sind in Programm 15.5 angegeben, wobei wir uns allerdings auf die einfacheren Varianten beschränken.

Programm 15.5 *Map-Filter-Reduce* auf Maps

```
STRUCTURE MapHofs = {
  FUN _*: (β → γ) → Map α β → Map α γ
  DEF f * m = f ∘ m
  FUN _*: (α × β → γ) → Map α β → Map α γ
  DEF f * (a ↦ b) = (a ↦ f(a, b))
  DEF f * (m1 ∪ m2) = (f * m1) ∪ (f * m2)
  FUN _▷: Map α β → (α → Bool) → Map α β
  FUN _▷: Map α β → (β → Bool) → Map α β
  FUN _▷: Map α β → (α × β → Bool) → Map α β
  DEF (a ↦ b) ▷ p = IF p(a) THEN (a ↦ b) ELSE □ FI
  DEF (a ↦ b) ▷ p = IF p(b) THEN (a ↦ b) ELSE □ FI
  DEF (a ↦ b) ▷ p = IF p(a, b) THEN (a ↦ b) ELSE □ FI
  DEF (m1 ∪ m2) ▷ p = (m1 ▷ p) ∪ (m2 ▷ p)        -- zu simpel!
      ...
  FUN _/: (β × β → β) → Map α β → β
  DEF ⊕/(a ↦ b) = b
  DEF ⊕/(m1 ∪ m2) = (⊕ / m1) ⊕ (⊕ / m2)
}
```

Da Maps nur spezielle Funktionen sind, ist der einfache „$*$“-Operator nichts anderes als die Funktionskomposition. Im Zusammenhang mit effizienter Speicherung ist diese Realisierung allerdings nicht empfehlenswert. Darauf kommen wir in Abschnitt 15.5 unter dem Stichwort „Memoization“ noch einmal zurück. Man beachte, dass bei $f * m$ natürlich f auch wieder eine Map sein kann.

Es gibt auch Applikationen, bei denen man eine Funktion auf alle Paare (a, b) mit $b = m\ a$ anwenden will. Diese Variante kann per Overloading ebenfalls mit $f * m$ geschrieben werden. (Wir geben die Definition nur für den Operator $\cup$ an, weil es bei $\overleftarrow{+}$ subtile technische Probleme wegen Definiertheitsfragen gibt; aber aus Programm 15.4 wissen wir, dass $\overleftarrow{+}$ auf $\cup$ zurückgeführt werden kann.)

Die Filter-Operation geben wir in drei überlagerten Varianten an, je nachdem ob wir den Definitionsbereich, den Wertebereich oder beide gemeinsam testen. Man beachte, dass wir es uns bei der letzten Definitionszeile von „$\triangleright$“ allzu leicht machen. Wir müssten die drei überlagerten Varianten in drei identischen Definitionen hinschreiben, die aber für den Compiler durch geeignete Typannotationen unterscheidbar gemacht werden.

Bei der Reduce-Operation geben wir nur die einfachste Variante an. Man beachte, dass diese auch nur für assoziative und kommutative Operatoren „$\oplus$“ sinnvoll ist, weil auf den Elementen einer Map keine Ordnung angenommen wird. Streng genommen müssten wir unsere Typisierung also durch die Angabe einer entsprechenden Typklasse präzisieren:

FUN $\_/:\ (\beta \times \beta \to \beta)$: $Assoc\ \&\ Comm \to Map\ \alpha\ \beta \to \beta$

Man beachte, dass eine Definition von Reduce, die nicht auf dem Operator $\cup$ basiert, sondern auf dem Operator $\overleftarrow{+}$, wesentlich schwieriger ist.

Für den Reduce-Operator bietet sich noch eine andere Möglichkeit an (die in vergleichbarer Weise für alle möglichen Arten von Datenstrukturen gilt). Wir führen eine spezielle Funktion ein, die die Map in eine lazy Liste von Paaren verwandelt:

FUN seq: $Map\ \alpha\ \beta \to List(\alpha \times \beta)$ $\quad -- \ List\ ist\ lazy$
DEF $seq\ \square = \Diamond$
DEF $seq((a \mapsto b) \cup m) = (a, b)\ \cdot{:}\ seq(m)$

Dann können wir alle Varianten von Reduce (und ähnlichen Operationen) direkt von Listen auf Maps übertragen. Durch die Laziness wird sichergestellt, dass die Liste nicht wirklich erzeugt und gespeichert wird. (Damit hat man de facto das realisiert, was z. B. in JAVA unter dem Stichwort *Iterator* firmiert.)

15.3.3 Prädikate höherer Ordnung auf Maps

Im Zusammenhang mit der Spezifikation und Entwicklung von komplexen Map-basierten Programmen (die wir gleich in Kapitel 16 diskutieren werden) braucht man auch Prädikate höherer Ordnung. Dann lassen sich Bedingungen formulieren, die gelten müssen, damit eine Map eine Lösung für ein gegebenes Problem repräsentiert. Programm 15.6 fasst die notwendigen Prädikate und deren Eigenschaften in einer Spezifikation *MapPredicates* zusammen.

Programm 15.6 Prädikate höherer Ordnung auf dem Datentyp Map

SPECIFICATION $MapPredicates = \{$
FUN $\boxdot:\ (\alpha \times \beta \to Bool) \to Map\ \alpha\ \beta \to Bool$ $\quad -- \ pointwise$
FUN $\boxdot_{dom}:\ (\alpha \to Bool) \to Map\ \alpha\ \beta \to Bool$ $\quad -- \ pointwise\ on\ domain$
FUN $\boxdot_{ran}:\ (\beta \to Bool) \to Map\ \alpha\ \beta \to Bool$ $\quad -- \ pointwise\ on\ range$
FUN $\boxtimes:\ (\alpha \times \beta \to \alpha \times \beta \to Bool) \to Map\ \alpha\ \beta \to Bool$ $\quad -- \ mutually$
FUN $\boxtimes_{dom}:\ (\alpha \to \alpha \to Bool) \to Map\ \alpha\ \beta \to Bool$ $\quad -- \ mutually\ on\ domain$
FUN $\boxtimes_{ran}:\ (\beta \to \beta \to Bool) \to Map\ \alpha\ \beta \to Bool$ $\quad -- \ mutually\ on\ range$
...
PROP $\boxdot\ p\ m = \forall a \in dom\ m \bullet p(a, m\ a)$
PROP $\boxdot_{dom}\ p\ m = \forall a \in dom\ m \bullet p(a)$
PROP $\boxdot_{ran}\ p\ m = \forall b \in ran\ m \bullet p(b)$
PROP $\boxtimes\ p\ m = \forall a_1 \in dom\ m,\ a_2 \in dom\ m,\ a_1 \neq a_2 \bullet p(a_1, m\ a_1)(a_2, m\ a_2)$
...
$\}$

Das Prädikat $\boxdot\ p\ m$ prüft, ob $p(a, b)$ für alle Elemente der Map m mit $m\ a = b$ gilt, ob p also „elementweise“ („*pointwise*“) für die Map m erfüllt ist.

Analog prüfen $\boxdot_{dom}\ p\ m$ und $\boxdot_{ran}\ p\ m$ die Gültigkeit von p für alle Elemente des Definitions- bzw. Wertebereichs. Diese Prädikate lassen sich direkt oder indirekt durch den Map- und den Reduce-Operator implementieren:

$$\boxdot\ p\ m = \wedge\ /\ p * m$$

Das Prädikat $\boxtimes\ p\ m$ ist wesentlich komplexer und aufwendiger: Es testet die Gültigkeit eines Prädikats p für je zwei Elemente $(a_1,\ m\ a_1)$ und $(a_2,\ m\ a_2)$ der Map, also „*pairwise*“ oder „*mutually*“. Auch hier gibt es analoge Varianten $\boxtimes_{dom}$ und $\boxtimes_{ran}$, die nur die Elemente des Definitions- oder des Wertebereichs betrachten.

Wir werden die Operatoren $\boxdot_{dom}$, $\boxdot_{ran}$, $\boxtimes_{dom}$ und $\boxtimes_{ran}$ per Overloading einfach als $\boxdot$ bzw. $\boxtimes$ schreiben, wenn die Typen der Prädikate eine eindeutige Erkennung zulassen.

Wir illustrieren den Gebrauch dieser Prädikate anhand der wohl bekannten Aufgabe des n-Damen-Problems. Dieses Problem lässt sich sehr kompakt mit Hilfe der Operatoren aus Programm 15.6 *spezifizieren*.

Beispiel 15.1 (Das n-Damen-Problem)

Es sollen n Damen auf einem $n \times n$-Schachbrett so platziert werden, dass keine Dame im Einflussbereich einer anderen steht. Dabei gelten die üblichen Schachregeln, nach denen eine Dame horizontal, vertikal und diagonal bewegt werden darf.

Das rechte Bild in Abbildung 15.1 zeigt eine Lösung des 4-Damen-Problems. Das linke Bild zeigt für eine der Damen, welche Felder aufgrund der Anforderung *friendly* aus der Spezifikation in Programm 15.7 für die anderen Damen blockiert werden.

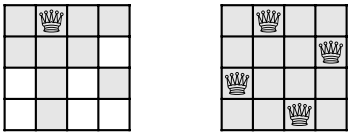


Abb. 15.1: Das 4-Damen-Problem

Programm 15.7 zeigt eine Spezifikation des n-Damen-Problems. Die Funktion *queens* bildet Mengen von Damen auf Mengen von Lösungen ab. Eine Lösung $m \in queens(QS)$ ist eine Map m, die alle Damen der Menge QS auf dem Schachbrett so platziert, dass sich deren Positionen in Zeilen, Spalten und Diagonalen nicht überschneiden. Diese Überschneidungsfreiheit wird durch

Programm 15.7 Spezifikation des n-Damen-Problems

```
FUN queens: Set Queen → Set(Map Queen Position)
PROP m ∈ queens(QS) ⟺ dom m = QS ∧ ⊠_ran friendly m

FUN friendly: Position → Position → Bool
PROP friendly p1 p2 ⟺    row p1 ≠ row p2          -- Zeilen
                       ∧ col p1 ≠ col p2          -- Spalten
                       ∧ up p1 ≠ up p2            -- Diagonalen
                       ∧ down p1 ≠ down p2        -- Diagonalen
```

das Prädikat *friendly* gesichert, das mit Hilfe von $\boxtimes_{ran}$ für je zwei Elemente der Ergebnis-Map überprüft wird.

Ein Algorithmus, der direkt aus dieser Spezifikation abgeleitet würde, wäre extrem ineffizient: Ausgehend von n Damen und $p = n * n$ Positionen würden zunächst p^n Maps erzeugt, die alle möglichen Positionierungen der n Damen repräsentieren (also auch die, bei denen mehrere Damen auf der gleichen Position stehen). Jede dieser Maps würde dann mit $\boxtimes$ *friendly* auf das „friedliche Nebeneinander" der Damen überprüft. Auf diese Weise probiert man alle potenziellen Zuordnungen durch und erhält die Menge der Lösungen. Dazu kommt, dass die Anwendung von $\boxtimes$ *friendly* auf eine einzelne Map auch noch relativ aufwendig ist: Das Prädikat *friendly* muss auf je zwei Damen angewendet werden, d. h. bei n Damen $\frac{n(n-1)}{2}$ mal; und *friendly* besteht selbst wiederum aus 4 Tests.

Dies zeigt, dass Programm 15.7 nur als *Spezifikation* der Aufgabenstellung brauchbar ist. Als *Implementierung* benötigen wir ein wesentlich effizienteres Programm. In Kapitel 16 werden wir zeigen, wie sich eine solche Implementierung aus unserer obigen Spezifikation formal gewinnen lässt.

15.4 Maps und Funktionen: Zwei Seiten einer Medaille

Wir wollen die Ähnlichkeit von Maps und Funktionen betonen, nicht ihre Unterschiede. Das soll auch in den Notationen sichtbar werden. Deshalb führen wir für den Aufbau von Maps neben den konstruktiven Operatoren aus Programm 15.3 auch eine kompakte Schreibweise im Stil der λ-Ausdrücke von klassischen Funktionen ein. Dies geschieht analog zur Vorgehensweise bei Arrays in Kapitel 14 und soll im Folgenden genauer betrachtet werden.

Trotz der Ähnlichkeit von Maps und Funktionen darf man die real existierenden Unterschiede nicht einfach ignorieren. Und der größte Unterschied ist sicher der, dass Maps nur über endlichen Mengen gebildet werden können. (Es müssen sogar „kleine" Mengen sein, die im Speicher eines Rechners Platz haben.) Deshalb wählen wir für die grundlegende Beschreibung von Maps ei-

ne Schreibweise, die sich subtil von den λ-Termen der Funktionsdefinitionen unterscheidet.

Definition (Map als λ-Term)

Sei $s\colon \mathit{Set}\ \alpha$ eine gegebene Menge und $f\colon \alpha \to \beta$ eine Funktion. Dann können wir folgende Map bilden:

VAL m: $\mathit{Map}\ \alpha\ \beta$
DEF $m = \lambda\, a \in s \bullet f(a)$

Im Gegensatz zu den üblichen λ-Termen, bei denen nur eine Typisierung der Art $\lambda\, a\colon T \bullet \ldots$ erfolgt, binden wir hier mittels $\lambda\, a \in s \bullet \ldots$ die Variable a an die Elemente der Menge s. Dabei wird gleichzeitig s als Definitionsmenge der Map m festgelegt.

Diese Notation entspricht der folgenden Map-Reduce-Konstruktion:

DEF $m = \cup\,/\ \mathit{singleton} * s$ WHERE $\mathit{singleton} = \lambda\, a \bullet (a \mapsto f(a))$

Hier wird zu jedem $a \in s$ die einelementige Map $(a \mapsto f(a))$ gebildet, und diese Maps werden dann mittels $\cup$ zu einer großen Map verschmolzen.

Das zeigt, wie eng die kompakte Beschreibung von Maps mit dem Map-Operator auf *Set* zusammenhängt (der seinerseits wieder verwandt ist mit den so genannten Iteratoren in Sprachen wie JAVA). Man könnte sich auch noch kompaktere Schreibweisen als $\lambda\, a \in s \bullet f(a)$ vorstellen, die ganz ohne Bezug zu den Individuen $a \in s$ auskommen. Naheliegend wäre z. B.

FUN $\_ \mid \_\colon (\alpha \to \beta) \times \mathit{Set}\ \alpha \to \mathit{Map}\ \alpha\ \beta$
DEF $f \mid s = \cup\,/\ \mathit{singleton} * s$ WHERE $\mathit{singleton} = \lambda\, a \bullet (a \mapsto f(a))$

Allerdings werden wir im Zusammenhang mit rekursiven Definitionen und Memoization nicht immer ohne Bezüge auf $a \in s$ auskommen. Deshalb behalten wir beide Notationen nebeneinander bei.

Anmerkung 1: Wie wir schon bei den Arrays angemerkt haben, fühlen sich manche Leute von einer solchen λ-artigen Notation irritiert. Sie wären entspannter, wenn man anstelle von $\lambda\, a \in s \bullet f(a)$ eine Spezialnotation wie $[\ a \mapsto f(a)\ \mid\ a \in s\ ]$ verwenden würde. Aber mit Spezialnotationen wollen wir lieber sparsam umgehen.

Anmerkung 2: Es wäre auch eine Variante von Maps denkbar, die beim Definitionsbereich den Unterschied zwischen Mengen und Typen nivelliert. Sei $s\colon \mathit{Set}\ \alpha$ gegeben. Dann führen wir den Subtyp TYPE $\alpha_s = (\alpha \mid \_ \in s)$ *ein und bilden Map $\alpha_s\ \beta$. Allerdings kann beim heutigen Stand der Compilertechnik nicht damit gerechnet werden, dass die hohe Dynamik dieser Konstruktion effizient implementiert werden kann.*

15.5 Maps, Funktionen und Memoization

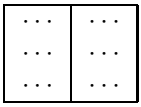
Wie wir gesehen haben, sind Maps spezielle, nämlich „eingefrorene“ Funktionen. Andererseits sind aber viele der Operatoren aus Programm 15.3 auch für

klassische Funktionen sinnvoll. So kann man z. B. einen Update-Operator für Funktionen definieren (sofern der Wertebereich zur Typklasse *Eq* gehört):

FUN $\_\,\_ \Leftarrow \_ \colon (\alpha \to \beta) \times \alpha \times \beta \to (\alpha \to \beta)$ VAR α: *Eq*
DEF $(f\ a \Leftarrow b)(a') =$ IF $a = a'$ THEN b ELSE $f(a')$ FI

Wenn man das allerdings mehrfach iteriert, dann ergibt sich eine lange und damit höchst ineffiziente IF-Kaskade. In diesem Falle wäre z. B. eine Hash-Tabelle erheblich effizienter.

Diese Beobachtungen – die Ähnlichkeit von Maps und Funktionen, sowie die Ineffizienz von naiven Update-Implementierungen – legen es nahe, beide Konzepte zu verschmelzen. Damit würden intern durch den Compiler sowohl Funktionen als auch Maps einheitlich dargestellt als Paare, bestehend aus einer reinen Map und einer reinen Funktion (d. h., einem λ-Term):

$f = ($  $, \lambda x \bullet \ldots)$ -- *Funktion als Paar (Map, λ-Term)*

Bei der Applikation $f(a)$ einer so implementierten Funktion wird zunächst in der Map nachgesehen, ob das Element a dort eingetragen ist. Falls nein, wird der λ-Term aufgerufen.

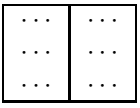
Die beiden Spezialfälle der reinen Funktionen und der reinen Maps fügen sich hier ganz einfach ein:

- „*Echte Funktionen*“ haben einen leeren Map-Anteil.
- „*Echte Maps*“ haben als Funktionsanteil die undefinierte Funktion $\lambda x \bullet \bot$.

Ein guter Compiler sollte in der Lage sein, das Vorliegen dieser Spezialfälle in den meisten Fällen zu erkennen und entsprechend zu optimieren.

Memoization

In der obigen integrierten Implementierung ist (bei Bedarf) das Prinzip der Memoization besonders einfach zu realisieren. Wir betrachten dazu den besonders anspruchsvollen Fall einer rekursiven Funktion. Diese hat eine Form der folgenden Bauart (mit einem Ausdruck $\mathcal{E}_x$ als Argument des rekursiven Aufrufs von f):

DEF$f = ($  $, \lambda x \bullet \ldots\ f(\mathcal{E}_x)\ \ldots)$ -- *rekursive Funktion*

Wenn bei einer Applikation $f(a)$ das Argument a nicht im Map-Teil eingetragen ist, wird der λ-Term aufgerufen, was gegebenenfalls zu einem rekursiven Aufruf $f(\mathcal{E}_a)$ führt, der seinerseits weitere rekursive Aufrufe auslösen kann. Sobald bei einem dieser Aufrufe das Ergebnis $y'_a = f(\mathcal{E}'_a)$ vorliegt, wird

das Paar $(\mathcal{E}'_a, y'_a)$ in die Map eingetragen und steht von da an bei allen weiteren Aufrufen zur Verfügung. Damit wird das Prinzip „Speichern statt neu Berechnen“ unmittelbar umgesetzt. Und weil diese rein interne Änderung semantiktreu ist, werden auch die Prinzipien der Funktionalen Programmierung nicht verletzt.

Es gibt nur eine Komplikation: Wann lohnt es sich, errechnete Ergebnisse zu speichern? Wenn ein Wertepaar $(\mathcal{E}'_a, y'_a)$ nur ein einziges Mal gebraucht wird, bedeutet das anschließende Speichern sowohl Zeit- als auch Platzverschwendung. Leider ist die Antwort auf diese Frage generell nicht entscheidbar und auch in speziellen Fällen zwar entscheidbar, aber nur mit gewaltigem Aufwand. Was das Ganze noch schlimmer macht: Das Aufblähen des Map-Teils mit Mengen von unnötigen Einträgen führt schließlich zu massiven Effizienzverlusten (anstelle der erhofften Optimierung).

Die Lösung dieses Dilemmas liegt letztlich wieder darin, die Entscheidung an den Programmierer zu delegieren. Er kann im Allgemeinen wissen, ob sich Memoization lohnt oder nicht. Alles, was er braucht, ist ein Sprachmittel, um seine Entscheidung zu formulieren. Deshalb sollte man – in Analogie zu Schlüsselwörtern wie LINEAR und LAZY – ein entsprechendes Schlüsselwort bereitstellen:

FUN f: $A \to B$ MEMOIZED

Man kann das weiter verfeinern, indem man noch ein Prädikat oder eine Menge angibt, um diejenigen Argumente zu charakterisieren, für die Memoization erfolgen soll:

FUN f: $A \to B$ MEMOIZED ON p −− *Prädikat* p: $A \to Bool$
FUN f: $A \to B$ MEMOIZED ON s −− *Menge* s: *Set* A

Man beachte aber, dass es in der Mengen-Variante noch immer zwei Möglichkeiten gibt:

- Man kann das übliche „Memoization-on-the-fly“ anwenden; das heißt, sobald ein Wert $f(a)$ mit $a \in s$ berechnet ist, wird er in die Map eingetragen.
- Man kann „Pre-Memoization“ anwenden; das heißt, beim ersten Aufruf von f wird die Map für die ganze Menge s bestimmt. (Dies entspricht der obigen Kompaktnotation $(f \mid s)$.)

Anmerkung: Die letzte dieser Varianten hat einen besonderen implementierungstechnischen Reiz. Weil die Menge s üblicherweise in der gleichen Art gespeichert ist wie die Map (normalerweise über Hash-Tabellen oder Rot-Schwarz-Bäume), kann man die Struktur beibehalten und nur an den Enden die Werte $f(a)$ eintragen.

16

Beispiel: Synthese von Programmen

Wir suchen niemals die Dinge, sondern das Suchen nach ihnen.

Blaise Pascal

Zum Abschluss des Bereichs „Datenstrukturen“ illustrieren wir die Verwendung von Maps zur Darstellung von Aufgaben und ihren Lösungsmengen. Unser Beispiel betten wir ein in die Anwendung der Funktionalen Programmierung zur Programmsynthese. (Bei der automatischen Programmsynthese leitet man Programme bzw. Algorithmen aus einer Spezifikation des Problems und Axiomen über das Hintergrundwissen ab.)

Mit der *Spezifikation* beschreiben wir *formal* den Anwendungsbereich und die Problemstellung. Wissen über algorithmische Grundlagen, mathematische Gesetze und Datenstrukturen bilden die *Axiome*, mit denen wir aus einer Spezifikation ein ausführbares Programm ableiten können. Wir können hierfür genau unsere Propertys aus Abschnitt 1.1.10 verwenden.

Da es sich meist um kompliziertere und aufwendigere Probleme handelt, geschieht die Programmableitung meistens halbautomatisch, also mit interaktiver Unterstützung durch den Benutzer, der Strategien zum Algorithmendesign anwendet und verfeinert.

Eine automatische oder wenigstens teilweise automatisierte Generierung von Implementierungen aus Spezifikationen erhöht die Zuverlässigkeit der erzeugten Programme und verringert gleichzeitig die Entwicklungszeit und damit die Herstellungskosten. Durch die Verwendung gezielter Strategien für die Programmsynthese kann man auch aus allgemein formulierten Basisalgorithmen effiziente Spezialalgorithmen ableiten.

Wir betrachten in diesem Kapitel ein besonders anspruchsvolles Beispiel: die Ableitung von Algorithmen zur *globalen Suche*. Dabei wird ein Suchraum, der eine Menge von Lösungskandidaten für ein Problem enthält, durch wiederholtes Aufteilen und Filtern eingeschränkt mit dem Ziel, konkrete Lösungen zu extrahieren. Wir werden sehen, dass sich Algorithmen für solche Probleme systematisch erzeugen lassen.

16.1 Globale Suche

Ziel der *globalen Suche* ist das Finden von Lösungen zu einem Problem, wobei von einer Menge von Lösungskandidaten, dem so genannten *Suchraum* ausgegangen wird. Dieser Suchraum wird durch Aufteilen und Filtern schrittweise verkleinert, bis man zu tatsächlichen Lösungen gelangt. Beim Verkleinern dürfen natürlich keine Lösungen verloren gehen.

Im Gegensatz dazu geht man bei Verfahren der *lokalen Suche*, wie Hillclimbing, Tabu Search oder Simulated Annealing [27], von konkreten (Teil-) Lösungen aus und untersucht deren lokale Nachbarschaft auf Verbesserungen, die dann wiederum den Ausgangspunkt für die weitere Suche darstellen.

Wenn wir Algorithmen zur globalen Suche beschreiben oder generieren wollen, müssen wir zunächst adäquate Mittel zur Darstellung von Suchräumen und deren Einschränkung bereitstellen. Diese können wir dann verwenden, um konkrete Probleme zu spezifizieren. Und aus diesen Spezifikationen leiten wir schließlich Programme ab.

16.1.1 Suchräume

Ein *Suchraum* für ein Problem ist eine Menge von Lösungskandidaten, die durch Bedingungen, die das Problem beschreiben, eingeschränkt wird.

Definition (Suchraum)

Wir repräsentieren einen Suchraum durch eine *Basismenge* S und ein *Constraint* C, das die Menge beschränkt bzw. *filtert*:[1]

$$S \triangleright C = \{x \in S \mid C(x)\}$$

Diese Darstellung kann auf verschiedene Weise interpretiert werden:

- *Semantische Sicht:* $S \triangleright C = \{\, x \in S \mid C(x)\}$ beschreibt die größte Teilmenge S' von S, deren Elemente x das Constraint C erfüllen.
- *Operationale Sicht:* $S \triangleright C$ bewirkt eine explizite Aufzählung aller Elemente von S und deren nachfolgende Filterung durch C.
- *Repräsentationssicht:* Hier betrachten wir $\triangleright$ als Typkonstruktor:
  TYPE $\mathit{ConstrainedSet} = \_ \triangleright \_ : (\mathit{base}: \mathit{Set}\ \alpha,\ \mathit{constraint}: \alpha \to \mathit{Bool})$

Hinweis: Die leere Menge kann entweder durch $\emptyset \triangleright C$ mit beliebigem Constraint C oder durch $S \triangleright \mathit{false}$ mit beliebiger Basismenge S dargestellt werden. In beiden Fällen schreiben wir vereinfachend $\emptyset$.

[1] Wir weichen in diesem Kapitel von unserer Konvention ab, Typen groß und Werte und Funktionen klein zu schreiben. Aus Gründen der besseren Lesbarkeit bezeichnen wir hier Datenstrukturen wie Maps und Sets mit Großbuchstaben und ihre Elemente mit Kleinbuchstaben.

Wir brauchen uns im Folgenden nicht vorzeitig auf eine der drei obigen Interpretation festzulegen, da die vorgestellten Methoden mit jeder dieser Sichtweisen umgehen können. Erst am Ende unserer Programmentwicklung wird es notwendig sein, sich für eine passende Repräsentation zu entscheiden. (Allerdings zielen manche Entwicklungsschritte schon in eine bestimmte Richtung.)

16.1.2 Einschränkung von Suchräumen

Suchräume können sehr groß und sogar unendlich sein. Wenn wir sie bei der globalen Suche schrittweise einschränken, kann das auf zwei Weisen geschehen:

- *Reduktion:* Wir eliminieren Elemente aus der Basismenge S, von denen wir wissen, dass sie C nicht erfüllen (s. Abbildung 16.1):

  Finde eine (bzgl. S kleinere) Menge S', so dass $S \triangleright C = S' \triangleright C$ gilt.

  Häufig kann man auch zu einem vereinfachten Constraint C' übergehen:

  Finde eine Menge S' und ein Constraint C' mit $S \triangleright C = S' \triangleright C'$.

Abb. 16.1: Reduktion eines Suchraums

- *Aufteilen (Splitting):* Der Suchraum wird in Teilräume zerlegt, die rekursiv durchsucht werden (s. Abbildung 16.2):

  Finde Mengen $S_1, \ldots, S_k$, so dass $S \triangleright C = (S_1 \triangleright C) \cup \ldots \cup (S_k \triangleright C)$ gilt.

  Häufig kann man auch hier zu vereinfachten Constraints $C_1, \ldots, C_k$ übergehen:

  Finde Mengen $S_1, \ldots, S_k$ und Constraints $C_1, \ldots, C_k$ mit
  $S \triangleright C = (S_1 \triangleright C_1) \cup \ldots \cup (S_k \triangleright C_k)$.

Abb. 16.2: Suchraumaufteilung

Beim Aufteilen eines Suchraums müssen die entstehenden Teilräume zwar nicht zwingend disjunkt sein, allerdings ist dies aus Effizienzgründen wünschenswert. Anderenfalls können Memoization-Techniken helfen, hinreichend schnelle Algorithmen zu erhalten.

Eine geschickte Aufteilung eines Suchraums erlaubt oft eine neue Reduktion seiner Teilräume. Im Extremfall kann man die Reduktion auch als Aufteilung des Suchraums in zwei Teilräume ansehen, wobei ein Teilraum ausschließlich ungültige Elemente enthält.

Die Reduktion entspricht dem Hinzufügen einer Constraint-Konjunktion zu C, wodurch die Lösungsmenge des Problems reduziert wird und eine weitere Suche effizienter verlaufen kann. Die Aufteilung eines Suchraums kann man hingegen als das Hinzufügen einer Constraint-Disjunktion betrachten. Die einzelnen Teilsuchräume müssen nachfolgend rekursiv durchsucht werden, wodurch der Berechnungsaufwand exponentiell anwächst. Daher ist es vorteilhaft, zunächst so viele Reduktionsschritte wie möglich auszuführen, bevor man eine Aufteilung des Suchraums vornimmt und die Teilräume rekursiv durchsucht. In der Literatur wird dieses Vorgehen, die deterministischen den nichtdeterministischen Berechnungen vorzuziehen, auch *Andorra-Prinzip* [144, 38] genannt.

16.1.3 Suchräume und partielle Lösungen

Neben der abstrakten Repräsentation von Suchräumen durch Mengen, wie wir sie verwenden, kann man Suchräume ebenso durch ihre *partiellen Lösungen* repräsentieren. Die partielle Konfiguration $[q_1 \mapsto (1,2),\ q_2 \mapsto (2,7)]$ des 8-Damen-Problems stellt so beispielsweise den Suchraum aller Lösungen mit der gegebenen Positionierung zweier Damen q_1 und q_2 dar. Fügt man eine dritte Dame $[q_3 \mapsto (3,3)]$ hinzu, so stellt dies einerseits eine Erweiterung der partiellen Lösung dar und auf der anderen Seite eine Reduktion des verbleibenden Suchraums. Gemäß dieser Dichotomie entspricht dann das Splitten eines Suchraums der Enumeration von Teillösungen.

Auch wenn beide Versionen äquivalent sind, ziehen wir für unseren konzeptuellen Ansatz die abstraktere Beschreibung mit Hilfe von Suchräumen der stückweisen Enumeration bzw. Vervollständigung von partiellen Lösungen vor.

16.1.4 Basisregeln für Suchräume

Als Basis zur Transformation von Programmen über Suchräumen, also beschränkten Mengen, stellen wir in Programm 16.1 eine Reihe von Regeln bereit. Dabei stützen wir uns auf einen Typ *Constraint*, dessen Details wir hier offen lassen, der aber im Wesentlichen Prädikaten ($\alpha \to \mathit{Bool}$) entspricht. Über diesen Constraints gibt es die üblichen Booleschen Operatoren wie z. B.

FUN _ ∧ _ : *Constraint* × *Constraint* → *Constraint* −− *Konjunktion*
FUN _ ∨ _ : *Constraint* × *Constraint* → *Constraint* −− *Disjunktion*
...

Außerdem verwenden wir den Typ *Set* α der Mengen über Elementen vom Typ α mit den üblichen Operationen Vereinigung, Durchschnitt etc. (vgl. Programm 10.9) sowie den üblichen Funktionen zur Programmierung nach dem *Map-Filter-Reduce*-Paradigma, die analog zu denen bei Sequenzen definiert sind (vgl. Abschnitt 1.2.2).

Programm 16.1 Basisregeln für Suchräume

SPECIFICATION *ConstraintRules* = {
  PROP C_1 IS $(S_1 \cup S_2) \triangleright C = (S_1 \triangleright C) \cup (S_2 \triangleright C)$
  PROP C_2 IS $S \triangleright (C_1 \wedge C_2) = (S \triangleright C_1) \triangleright C_2$
  PROP C_3 IS $S \triangleright (C_1 \vee C_2) = (S \triangleright C_1) \cup (S \triangleright C_2)$
}

Wir verwenden die Notation „PROP *Name* IS *Property*“, um Propertys, also Eigenschaften von Operationen, Namen zu geben, auf die wir uns bei der Programmableitung beziehen können.

16.2 Problemlösungen als *Maps*

Probleme, wie wir sie hier mit globaler Suche behandeln wollen – kurz: *Suchprobleme* –, lassen sich im Allgemeinen durch eine Menge von Variablen beschreiben. Eine *Lösung* des Problems besteht dann aus einer Zuordnung von Werten zu diesen Variablen, die gewissen Bedingungen, d. h. der Problembeschreibung genügen. Solche Zuordnungen lassen sich sehr gut mit Hilfe von Maps darstellen.

Eine zentrale Rolle bei der Beschreibung von Suchproblemen spielen die Map-Prädikate der Spezifikation *MapPredicates*, die wir in Kapitel 15 in Programm 15.6 eingeführt haben.

16.2.1 Beispiel: Das n-Damen-Problem

Um die Diskussion anschaulicher zu machen, verwenden wir zur Illustration wieder das wohl bekannte n-Damen-Problem. Allerdings beginnen wir jetzt nicht mit der Fassung von Programm 15.7 aus Abschnitt 15.3.3, sondern nehmen gleich eine leichte Optimierung vor.

Wir wissen, dass eine konfliktfreie Anordnung von n Damen auf einem $n \times n$-Schachbrett nur möglich ist, wenn sich die Damen auf verschiedenen

Zeilen befinden. Daher können wir von vornherein Zeilen und Damen einander zuordnen. Wir verlieren dadurch keine Lösungen, da sich die Damen nicht durch spezielle Eigenschaften auszeichnen und damit potenziell austauschbar sind. Programm 16.2 zeigt eine optimierte Spezifikation, die auf dieser Idee beruht. Zur leichteren Lesbarkeit verwenden wir hier die (überlagerte) Schreibweise $\boxtimes\ friendly\ m$ anstelle der Form $\boxtimes_{ran}\ friendly\ m$.

Programm 16.2 Optimierte Spezifikation des n-Damen-Problems

FUN $queens$: $Set\ Queen \to Set(Map\ Queen\ Position)$
PROP $m \in queens(Q) \iff dom\ m = Q\ \wedge\ \boxdot\ onRow\ m\ \wedge\ \boxtimes\ friendly\ m$

FUN $onRow$: $Queen \times Position \to Bool$
DEF $onRow(queen, pos) = (row(pos) = index(queen))$

FUN $friendly$: $Position \to Position \to Bool$
DEF $friendly\ pos_1\ pos_2\ =\ col\ pos_1 \neq col\ pos_2$ $\quad -- Spalten$
$\wedge\ up\ pos_1 \neq up\ pos_2$ $\quad -- Diagonalen$
$\wedge\ down\ pos_1 \neq down\ pos_2$ $\quad -- Diagonalen$

Ein Algorithmus auf Basis dieser optimierten Spezifikation ist günstiger als die ursprüngliche Form in Programm 15.7. Zwar würden auch jetzt noch initial alle möglichen Zuordnungen der Damen auf dem Brett erzeugt, von diesen aber sofort eine größere Anzahl durch das „billige“ Prädikat $\boxdot\ onRow$ ausgeschlossen. Die Prüfung mit dem „teuren“ Prädikat $\boxtimes\ friendly$ erfolgt erst ganz am Schluss für die übrig gebliebenen n^n Alternativen und damit für eine deutlich geringere Anzahl von Lösungskandidaten als zuvor.

Ziel: Ein effizienter Algorithmus

Aber auch dies läßt sich natürlich noch optimieren. Statt einer einmaligen vollständigen Enumeration mit anschließender Suchraumeinschränkung können wir die Damen auch schrittweise platzieren und dabei jede neue Platzierung zu einer erneuten Suchraumeinschränkung nutzen. Abbildung 16.3 illustriert dies für das 8-Damen-Problem.

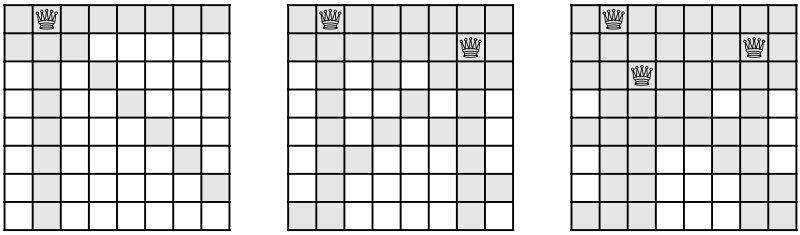


Abb. 16.3: Schrittweise Suchraumeinschränkung durch die Platzierung von Damen

Ausgehend von der schon optimierten Spezifikation hätte man auf einem leeren Schachbrett für jede Dame zunächst n=8 Möglichkeiten, sie zu platzieren und damit für unser 8-Damen-Problem noch einen recht großen Suchraum. Haben wir – wie im ersten Bild von Abbildung 16.3 – hingegen schon eine Dame in die erste Zeile positioniert, dann sind dadurch die grau markierten Felder für die weiteren Damen blockiert und somit der Suchraum eingeschränkt.

Dame Nummer 2 wird nun an eine der noch möglichen Positionen in Zeile 2 gesetzt und schränkt dadurch den verbleibenden Suchraum erneut ein.[2] Genauso reduziert jede weitere Dame den Suchraum und damit den Aufwand zur Platzierung der restlichen Damen. Auch die Kompatibilität ($\boxtimes$ *friendly*) der Positionierung muss immer nur für die jeweils neu gesetzte Dame geprüft werden und nicht für die bereits vorhandenen Damen untereinander.

Unser finales Ziel ist daher ein Algorithmus, der den Suchraum wie skizziert schrittweise einschränkt und damit den Suchaufwand im Vergleich zu den naiven Lösungen deutlich verringert.

16.2.2 Suchräume als Mengen von Maps

Bisher haben wir gezeigt, wie man mit Maps Lösungen von Suchproblemen darstellen kann. Wenn wir nicht nur einzelne Lösungen, sondern Lösungsmengen oder Suchräume repräsentieren wollen, müssen wir von Maps zu Mengen von Maps übergehen. Das haben wir im Grunde auch schon am Typ der Funktion *queens* in den Spezifikationen für das n-Damen-Problem gesehen.

Festlegung (Suchraum)

Im Folgenden verwenden wir den Begriff *Suchraum* speziell für Mengen von Maps, also für den Typ $Set(Map\ \alpha\ \beta)$.

Durch Prädikate auf Maps haben wir Bedingungen für Lösungen formuliert. Wenn wir die Anwendung dieser Prädikate auf Suchräume übertragen, dann können wir damit die Einschränkung der Suchräume beschreiben. Und das werden wir nutzen, um schließlich aus unseren Spezifikationen konkrete (und effiziente) Programme abzuleiten.

Wir beginnen mit der Repräsentation von Suchräumen als Mengen von Maps. Programm 16.3 gibt eine entsprechende Spezifikation *MapSpaces* an. Mit $[\![A \rightarrowtail B]\!]$ und $[\![A \rightarrow B]\!]$ bezeichnen wir die Menge der totalen bzw. partiellen Maps vom Definitionsbereich A in den Wertebereich B.

Die Komposition $\mathcal{M}_1 \otimes \mathcal{M}_2$ von Suchräumen definieren wir auf der Basis der Komposition jeder Map der Menge $\mathcal{M}_1$ mit jeder Map der Menge $\mathcal{M}_2$ durch die Operation $\cup$ (s. Programm 15.3). Entsprechend der Definition von

[2] Man könnte sich auch Strategien überlegen, bei denen man die Zeilen nicht in der Reihenfolge $1, 2, \ldots, n$ besetzt, sondern verteilter, weil so früher mehr Felder blockiert werden.

Programm 16.3 Mengen von Maps

```
SPECIFICATION MapSpaces[α, β] = {
  FUN ⟦ _ → _ ⟧ : Set α × Set β → Set(Map α β)          -- partielle Maps
  FUN ⟦ _ ↣ _ ⟧ : Set α × Set β → Set(Map α β)          -- totale Maps
  FUN _ ⊗ _ : Set(Map α β) × Set(Map α β) → Set(Map α β)
  PROP M ∈ ⟦ A → B ⟧ ⟺ (dom M ⊆ A) ∧ (ran M ⊆ B)
  PROP M ∈ ⟦ A ↣ B ⟧ ⟺ (dom M = A) ∧ (ran M ⊆ B)
  PROP M ∈ M1 ⊗ M2 ⟺ ∃M1 ∈ M1, ∃M2 ∈ M2 • M = M1 ∪ M2
  THM A1 IS ⟦ A1 ⊎ A2 ↣ B ⟧ = ⟦ A1 ↣ B ⟧ ⊗ ⟦ A2 ↣ B ⟧
  THM A2 IS ⟦ {a} ↣ B1 ∪ B2 ⟧ = ⟦ {a} ↣ B1 ⟧ ∪ ⟦ {a} ↣ B2 ⟧
  THM A3 IS M1 ⊗ (M2 ∪ M3) = (M1 ⊗ M2) ∪ (M1 ⊗ M3)
}
```

∪ werden nur „kompatible“ Maps vereint, also Maps, die auf dem Durchschnitt ihrer Definitionsbereiche übereinstimmen.

Zusätzlich zu den Propertys haben wir hier *Theoreme* angegeben. Diese sind Folgerungen, die sich aus den Propertys ableiten lassen. (Unsere Propertys entsprechen also dem, was man in der Logik Axiome nennt.) Die Theoreme *A1* bis *A3* beschreiben Eigenschaften der Komposition und Vereinigung von Suchräumen. Die Theoreme sind für totale genauso wie für partielle Maps anwendbar; wir zeigen hier jeweils die Variante für totale Maps.

Wir verwenden die Kurznotation $A_1 \uplus A_2$ um auszudrücken, dass zwei *disjunkte* Mengen vereinigt werden.

16.2.3 Constraints auf Mengen von Maps

Wir übertragen nun die Prädikate der Spezifikation *MapPredicates* aus Programm 15.6 auf Mengen von Maps und können so die Reduktion von Suchräumen beschreiben. Programm 16.4 gibt die Spezifikation von Constraints zur Suchraumeinschränkung an.

Theorem *B1* gilt nur für *partielle* Maps; denn bei totalen Maps werden auf der linken Seite sämtliche Maps eliminiert, sobald auch nur ein $a \in A$ das Prädikat p verletzt. Die Theoreme *B2* bis *B5* sind dagegen jeweils für partielle *und* totale Maps anwendbar. Der Kürze halber haben wir sie nur in den Versionen für totale Maps angegeben. Bei den Theoremen *B6* und *B7* muss der zweite Teilsuchraum genau die Singleton-Map $(a \mapsto b)$ enthalten. Das ist automatisch eine totale Map.

Die Theoreme *B1* bis *B3* treffen Aussagen über die Einschränkung von Suchräumen durch Prädikate über den Definitions- und Wertebereichen und den Elementen von Maps.

Theorem *B4* besagt, dass man Suchräume genauso vor wie nach ihrer Komposition filtern kann, solange dies „lokale“ Prädikate betrifft, also Eigen-

Programm 16.4 Constraints auf Suchräumen

$$
\begin{array}{l}
\textsc{specification}\ \mathit{MapConstraints}[\alpha,\beta] = \{ \\
\quad \textsc{thm}\ \mathit{B1}\ \textsc{is}\ [\![\, A \to B \,]\!] \triangleright (\boxdot_{dom}\, p) = [\![\, (A \triangleright p) \to B \,]\!] \qquad -\!-\ \mathit{nur\ für}\ \to \\
\quad \textsc{thm}\ \mathit{B2}\ \textsc{is}\ [\![\, A \rightarrowtail B \,]\!] \triangleright (\boxdot_{ran}\, q) = [\![\, A \rightarrowtail (B \triangleright q) \,]\!] \qquad -\!-\ \mathit{auch\ für}\ \to \\
\quad \textsc{thm}\ \mathit{B3}\ \textsc{is}\ [\![\, \{a\} \rightarrowtail B \,]\!] \triangleright (\boxdot\, p) = [\![\, \{a\} \rightarrowtail (B \triangleright p(a, \_)) \,]\!] \qquad -\!-\ \mathit{auch\ für}\ \to \\
\quad \textsc{thm}\ \mathit{B4}\ \textsc{is}\ (\mathcal{M}_1 \otimes \mathcal{M}_2) \triangleright (\boxdot\, p) = (\mathcal{M}_1 \triangleright (\boxdot\, p)) \otimes (\mathcal{M}_2 \triangleright (\boxdot\, p)) \ -\!-\ \mathit{auch\ für}\ \to \\
\quad \textsc{thm}\ \mathit{B5}\ \textsc{is}\ (\mathcal{M}_1 \otimes \mathcal{M}_2) \triangleright (\boxtimes\, p) = ((\mathcal{M}_1 \triangleright (\boxtimes\, p)) \otimes \mathcal{M}_2) \triangleright (\boxtimes\, p) \ -\!-\ \mathit{auch\ für}\ \to \\
\quad \textsc{thm}\ \mathit{B6}\ \textsc{is}\ (\mathcal{M} \otimes \{a \mapsto b\}) \triangleright (\boxtimes\, p) = ((\mathcal{M} \triangleright (\boxtimes\, p)) \triangleright (\boxdot\, p(a,b))) \otimes \{a \mapsto b\} \\
\quad \textsc{thm}\ \mathit{B7}\ \textsc{is}\ (\mathcal{M} \otimes \{a \mapsto b\}) \triangleright\ (\boxtimes_{ran}\, p) \\
\qquad\qquad\qquad\qquad = ((\mathcal{M} \triangleright (\boxtimes_{ran}\, p)) \triangleright (\boxdot_{ran}\, p(b))) \otimes \{a \mapsto b\} \\
\}
\end{array}
$$

schaften, die punktweise für jedes Element einer Map einzeln geprüft werden können. Dieses Theorem kann analog auf $\boxdot_{dom}$ und $\boxdot_{ran}$ übertragen werden.

Für $\boxtimes$ gilt das nicht, weshalb Theorem *B5* komplexer ist: Wenn wir einen Suchraum mit $(\boxtimes\, p)$ gefiltert haben und ihn durch Komposition erweitern, dann können die neu entstandenen Maps wieder Punktekombinationen enthalten, die p verletzen. Daher müssen wir das Ergebnis der Komposition erneut mit $(\boxtimes\, p)$ filtern.

Bei Theorem *B6* filtern wir ebenfalls die Komposition zweier Suchräume mit einem Prädikat p, das auf je zwei Punkte der Maps angewendet wird. Hierbei betrachten wir jetzt aber den Spezialfall, dass der zweite Suchraum nur die Map $(a \mapsto b)$ enthält. Statt den kombinierten Suchraum mit p zu filtern, können wir nun genauso gut zunächst den ersten Suchraum $\mathcal{M}$ bzgl. $(\boxtimes\, p)$ filtern und danach die Maps aus $\mathcal{M}$ punktweise auf Kompatibilität mit dem Paar (a, b) der einelementigen Map prüfen. Das Ergebnis wird schließlich mit dem einelementigen Suchraum $\{a \mapsto b\}$ komponiert (mit dem sich jetzt alle verbliebenen Maps vertragen).

Theorem *B7* ist eine Version von Theorem *B6*, bei der p nur auf den Wertebereichen der Maps arbeitet. Entsprechend verwenden wir hier die Filterprädikate $\boxtimes_{ran}$ und $\boxdot_{ran}$.

Durch Anwendung der letzten beiden Theoreme lassen sich oft entscheidende Effizienzgewinne erzielen: Die aufwendige Prüfung mit $(\boxtimes\, p)$ oder $(\boxtimes_{ran}\, p)$ wird auf ein schrittweises Filtern mit den weniger aufwendigen Prädikaten $(\boxdot\, p(a,b))$ und $(\boxdot_{ran}\, p(b))$ zurückgeführt. Wir werden diese Idee gleich benutzen, um einen effizienten Algorithmus für das n-Damen-Problem aus unserem einfachen Basisalgorithmus zu gewinnen.

16.3 Programmableitung

Jetzt wollen wir mit Hilfe unserer Theoreme und Propertys einen Algorithmus zur globalen Suche ableiten und optimieren. Wir arbeiten dabei zunächst

mit unserem Beispiel, dem n-Damen-Problem. Im zweiten Schritt verallgemeinern wir dieses Vorgehen und erhalten so einen generischen Algorithmus zur globalen Suche.

16.3.1 Das n-Damen-Problem – Von der Spezifikation zum Algorithmus

Wir starten mit der optimierten Spezifikation aus Programm 16.2. Für die gesuchte Funktion *queens* gilt offenbar (wobei wir mit *Pos* die Menge aller Positionen *pos*: *Position* bezeichnen):

$$\text{DEF } queens(Q) = [\![\, Q \rightarrowtail Pos \,]\!] \triangleright ((\boxdot\, onRow) \wedge (\boxtimes\, friendly))$$

Wenn wir die operationale Sichtweise zugrunde legen, dann ist das sogar schon eine primitive Definition der gesuchten Funktion: Alle Elemente des Suchraums, also alle möglichen Konfigurationen, werden explizit aufgezählt und gefiltert.

Wir leiten jetzt aus dieser initialen Definition durch Induktion über die Menge Q der Damen und passende Transformationen zunächst eine rekursive Definition für die Funktion *queens* her, die wir später weiter optimieren werden. Abbildung 16.4 zeigt die einzelnen Transformationsschritte.

$$queens(\emptyset) = \{\square\}$$

$$queens(Q \uplus \{q\}) = [\![\, Q \uplus \{q\} \rightarrowtail Pos \,]\!] \triangleright ((\boxdot\, onRow) \wedge (\boxtimes\, friendly))$$

$$\overset{A1}{\equiv} ([\![\, Q \rightarrowtail Pos \,]\!] \otimes [\![\, \{q\} \rightarrowtail Pos \,]\!]) \triangleright ((\boxdot\, onRow) \wedge (\boxtimes\, friendly))$$

$$\overset{C2}{\equiv} (([\![\, Q \rightarrowtail Pos \,]\!] \otimes [\![\, \{q\} \rightarrowtail Pos \,]\!]) \triangleright (\boxdot\, onRow)) \triangleright (\boxtimes\, friendly)$$

$$\overset{B4}{\equiv} (([\![\, Q \rightarrowtail Pos \,]\!] \triangleright (\boxdot\, onRow)) \otimes ([\![\, \{q\} \rightarrowtail Pos \,]\!] \triangleright (\boxdot\, onRow))) \triangleright (\boxtimes\, friendly)$$

$$\overset{B3}{\equiv} (([\![\, Q \rightarrowtail Pos \,]\!] \triangleright (\boxdot\, onRow)) \otimes [\![\, \{q\} \rightarrowtail Pos \triangleright onRow(q, \_) \,]\!]) \triangleright (\boxtimes\, friendly)$$

$$\overset{Def}{\equiv} (([\![\, Q \rightarrowtail Pos \,]\!] \triangleright (\boxdot\, onRow)) \otimes [\![\, \{q\} \rightarrowtail Pos_{(q,\_)} \,]\!]) \triangleright (\boxtimes\, friendly)$$

$$\overset{B5}{\equiv} ((([\![\, Q \rightarrowtail Pos \,]\!] \triangleright (\boxdot\, onRow)) \triangleright (\boxtimes\, friendly)) \otimes [\![\, \{q\} \rightarrowtail Pos_{(q,\_)} \,]\!]) \triangleright (\boxtimes\, friendly)$$

$$\overset{C2}{\equiv} ((([\![\, Q \rightarrowtail Pos \,]\!] \triangleright ((\boxdot\, onRow) \wedge (\boxtimes\, friendly))) \otimes [\![\, \{q\} \rightarrowtail Pos_{(q,\_)} \,]\!]) \triangleright (\boxtimes\, friendly)$$

$$\overset{Ind}{\equiv} (queens(Q) \otimes [\![\, \{q\} \rightarrowtail Pos_{(q,\_)} \,]\!]) \triangleright (\boxtimes\, friendly)$$

Abb. 16.4: Ableitung einer rekursiven Definition für das n-Damen-Problem

Haben wir eine leere Menge von Queens, so erhalten wir eine Menge, die nur die leere Map enthält.

Im anderen Fall zerlegen wir unseren Suchraum in zwei Teil(such)räume (Theorem *A1*) und transformieren den entstehenden Ausdruck dann schrittweise, indem die Constraints jeweils auf die Teilräume angewendet werden.

Mit Property *C2* bilden wir aus der Konjunktion von Bedingungen zwei nacheinander anzuwendende Filter. Den ersten, „inneren" Filter ($\boxdot\, onRow$) können wir dann nach Theorem *B4* in unabhängigen Schritten auf die Teilräume anwenden. Theorem *B3* reduziert den Suchraum $[\![\, \{q\} \rightarrowtail Pos \,]\!]$ zum Suchraum $[\![\, \{q\} \rightarrowtail Pos_{(q,\,\_)} \,]\!]$, der alle Abbildungen von q auf Positionen enthält, deren Zeilen gleich dem Index der Dame q sind:

$$Pos_{(q,\,\_)} \stackrel{Def}{=} Pos \triangleright onRow(q,\,\_) = \{p \in Pos \mid row(p) = index(q)\}$$

Mit Theorem *B5* dürfen wir ($\boxtimes friendly$) schließlich zuerst auf den ersten Teilraum anwenden, bevor das Ergebnis mit $[\![\, \{q\} \rightarrowtail Pos_{(q,\,\_)} \,]\!]$ komponiert und erneut bzgl. ($\boxtimes friendly$) gefiltert wird.

Per Induktion erhalten wir so eine rekursive Definition für die Funktion *queens*, die in Programm 16.5 angegeben ist. Um eine Lösung für $n+1$ Damen

Programm 16.5 Die Basislösung des n-Damen-Problems

DEF $queens(\emptyset) = \{\square\}$
DEF $queens(Q \uplus \{q\}) = (queens(Q) \otimes [\![\, \{q\} \rightarrowtail Pos_{(q,\,\_)} \,]\!]) \triangleright (\boxtimes friendly)$

$Q \uplus q$ zu berechnen, ermitteln wir zunächst die Lösungen für $queens(Q)$. Wir „erweitern" diese um die Dame q, indem wir aus jeder Map der bisherigen Lösungsmenge $\mid Pos_{(q,\,\_)} \mid$ neue Maps bilden. Dazu ergänzen wir jede einzelne Map um je einen Eintrag, der q auf eine der Positionen aus $Pos_{(q,\,\_)}$ abbildet. Diesen neuen Suchraum müssen wir am Ende wieder auf die Kompatibilität aller Damen prüfen und die Menge der potenziellen Lösungen entsprechend einschränken.

Man beachte: Während wir die Damen schrittweise hinzunehmen, ist die Menge der Positionen, d. h. die Größe des Schachbretts, von Anfang an festgelegt. Wenn wir also z. B. das 8-Damen-Problem lösen, dann berechnen wir unterwegs zwar *queens* für 7, 6, ..., 0 Damen. Wir lösen dabei aber *nicht* das 7-Damen- oder 6-Damen-Problem, da die Damen jeweils auf einem 8×8-Schachbrett positioniert werden.

Von „$\boxtimes$" zu „$\boxdot$"

Die eben abgeleitete rekursive Definition zur Lösung des n-Damen-Problems ist leider nicht besonders effizient. Ausgehend von einer Lösung $queens(Q)$ wird eine weitere Dame q hinzugefügt und dann die Kompatibilität aller Damen jeder neuen potenziellen Lösung untereinander geprüft.

Die Damen in Q sind auf dem Schachbrett aber schon so platziert, dass keine im Einflussbereich einer anderen steht; sonst wäre $queens(Q)$ keine Lösung. Es würde daher ausreichen, die Kompatibilität der Damen aus Q nur jeweils bzgl. der Position der neuen Dame q zu überprüfen und nicht noch einmal ihre Kompatibilität untereinander.

Der Schlüssel zu einer solchen Lösung ist das Theorem *B7*. Abbildung 16.5 zeigt die Ableitung des Programms. Wir beginnen mit der bisherigen rekursiven Definition der Funktion *queens*.

$$
\begin{aligned}
& queens(Q \uplus \{q\}) = (queens(Q) \otimes [\![\, \{q\} \rightarrowtail Pos_{(q,\_)} \,]\!]) \triangleright (\boxtimes\, friendly) \\
& \overset{A2}{\equiv} (queens(Q) \otimes \textstyle\bigcup_{p \in Pos_{(q,\_)}} \{q \mapsto p\}) \triangleright (\boxtimes\, friendly) \\
& \overset{A3}{\equiv} (\textstyle\bigcup_{p \in Pos_{(q,\_)}} queens(Q) \otimes \{q \mapsto p\}) \triangleright (\boxtimes\, friendly) \\
& \overset{C1}{\equiv} \textstyle\bigcup_{p \in Pos_{(q,\_)}} ((queens(Q) \otimes \{q \mapsto p\}) \triangleright (\boxtimes\, friendly)) \\
& \overset{B7}{\equiv} \textstyle\bigcup_{p \in Pos_{(q,\_)}} (((queens(Q) \triangleright (\boxtimes\, friendly)) \triangleright (\boxdot\, friendly(p))) \otimes \{q \mapsto p\}) \\
& \overset{Def}{\equiv} \textstyle\bigcup_{p \in Pos_{(q,\_)}} ((queens(Q) \triangleright (\boxdot\, friendly(p))) \otimes \{q \mapsto p\})
\end{aligned}
$$

Abb. 16.5: Optimierung: Von $\boxtimes$ zu $\boxdot$

Damit wir Theorem *B7* anwenden können, müssen wir unseren Ausdruck zunächst in eine entsprechende Form transformieren. Der Suchraum $[\![\, \{q\} \rightarrowtail Pos_{(q,\_)} \,]\!]$ enthält eine Menge von einelementigen Maps. Daher können wir ihn gemäß Theorem *A2* aufspalten, indem wir den Wertebereich $Pos_{(q,\_)}$ in einelementige Mengen zerlegen. Mit Theorem *A3* verteilen wir die Komposition über die Elemente der Vereinigung. Und nach Basisregel *C1* können wir statt des gesamten Suchraums auch seine Teilsuchräume filtern und die Ergebnismengen vereinigen.

Jetzt hat unser Ausdruck eine Form, in der wir Theorem *B7* anwenden können. Beachte, dass wir zur leichteren Lesbarkeit hier die (überlagerte) Schreibweise $(\boxtimes\, friendly)$ anstelle der Form $(\boxtimes_{ran}\, friendly)$ verwenden.[3] Ebenso schreiben wir beim Resultat jetzt auch $(\boxdot\, friendly(p))$ anstelle von $(\boxdot_{ran}\, friendly(p))$.

Das Ergebnis können wir weiter vereinfachen, da für $queens(Q)$ bereits per Definition $(\boxtimes\, friendly)$ gilt. Somit können wir diesen Filterausdruck weglassen und erhalten eine rekursive Definition, die statt auf dem aufwendigen Prädikat $\boxtimes$ nun nur noch auf $\boxdot$ beruht. Programm 16.6 zeigt die optimierte *queens*-Funktion.

[3] Hat man es tatsächlich mit $(\boxtimes\, p)$ zu tun, dann kann man in analoger Weise Theorem *B6* nutzen.

Programm 16.6 Eine effiziente Lösung des n-Damen-Problems

```
FUN queens: Set Queen → Set(Map Queen Position)
DEF queens(∅) = {□}
DEF queens(Q ⊎ {q}) =
    LET QS = queens(Q)
        f(pos) = (QS ▷ (⊡ friendly(pos))) ⊗ {q ↦ pos}
    IN  ∪ / (f ∗ Pos_(q, _))
```

Die Menge der zulässigen Positionierungen der Damen berechnen wir jetzt, indem wir schrittweise zu einer bestehenden Teillösung jeweils eine neue Dame auf dem Brett platzieren, deren Position mit denen der Damen der schon bestehenden Teillösung kompatibel ist. Das setzt genau das Vorgehen aus Abbildung 16.3 um.

Wie bisher bezeichnen $\uplus$ und $\cup$ die Vereinigung von Mengen, wobei im ersteren Fall die Argumentmengen disjunkt sind. Unsere Funktion *queens* macht intensiven Gebrauch von Funktionen höherer Ordnung: Zunächst wenden wir mit Hilfe der Map-Funktion $*$ die Funktion f auf alle Elemente der Menge $Pos_{(q,\_)}$ an. Wir erhalten eine Menge von Suchräumen, also eine Menge von Mengen von Maps, die wir dann mit Reduce zusammenfassen. Durch das Herausziehen (*invariant code motion*) von $queens(Q)$ aus dem Reduce-Ausdruck braucht dieser von p und q unabhängige Term nur einmal ausgewertet zu werden und wir erhalten einen effizienten Algorithmus für unser n-Damen-Problem.

Von Mengenfiltern zu akkumulierten Prädikaten

Die Implementierung aus Programm 16.6 können wir aber auch noch auf eine andere Art und Weise verbessern. Statt den aktuellen Suchraum jeweils bei Hinzunahme einer neuen Dame bzgl. seiner Kompatibilität mit jeder möglichen Platzierung der Dame zu filtern, gehen wir einfach umgekehrt vor: Wir sammeln die Kompatibilitätsconstraints in einem Akkumulationsparameter als Konjunktion auf und wenden diese Konjunktion dann jeweils nur auf die Position der neuen Dame an.

Dazu definieren wir eine Funktion *queens′* die einen zusätzlichen Akkumulationsparameter c für die Constraints erhält:

```
DEF queens′(Q, c) = queens(Q) ▷ (⊡ c)
```

Wir interpretieren „ ▷ “ jetzt als Typkonstruktor. Dann können wir durch eine einfache Transformation die Funktion *queens′*, die in Programm 16.7 angegeben ist, ableiten. Die Herleitung zeigt Abbildung 16.6.

Mit Programm 16.7 sind wir von einer abstrakten Beschreibung des Suchraums zu einer implementierungsnahen Beschreibung übergegangen und haben das Filtern des Suchraums durch das Aufsammeln von Constraints ersetzt,

$$
\begin{aligned}
&queens'(\emptyset,\ c) = \{\Box\}\\
&queens'(Q \uplus \{q\},\ c)\\
&= (\bigcup_{p \in Pos_{(q,\ \_)}} ((queens(Q) \triangleright (\boxdot\, friendly(p))) \otimes \{q \mapsto p\})) \triangleright (\boxdot\, c)\\
&\overset{C1}{\equiv} \bigcup_{p \in Pos_{(q,\ \_)}} (((queens(Q) \triangleright (\boxdot\, friendly(p))) \otimes \{q \mapsto p\}) \triangleright (\boxdot\, c))\\
&\overset{B4}{\equiv} \bigcup_{p \in Pos_{(q,\ \_)}} (((queens(Q) \triangleright (\boxdot\, friendly(p))) \triangleright (\boxdot\, c)) \otimes (\{q \mapsto p\} \triangleright (\boxdot\, c)))\\
&\overset{C2}{\equiv} \bigcup_{p \in Pos_{(q,\ \_)}} ((queens(Q) \triangleright (\boxdot\, (friendly(p) \wedge c))) \otimes (\{q \mapsto p\} \triangleright (\boxdot\, c)))\\
&\overset{Ind}{\equiv} \bigcup_{p \in Pos_{(q,\ \_)}} (queens'(Q, friendly(p) \wedge c) \otimes (\{q \mapsto p\} \triangleright (\boxdot\, c)))
\end{aligned}
$$

Abb. 16.6: Von Mengenfiltern zu akkumulierten Prädikaten

so dass jede Erweiterung der Teillösungen bzgl. der akkumulierten Constraints überprüft wird.

Programm 16.7 Das n-Damen-Problem: Akkumulation von Constraints

```
DEF queens(Q) = queens'(Q, λ _ • true)
FUN queens': (Set Queen) × (Position → Bool) → Set(Map Queen Position)
DEF queens'(∅, c) = {□}
DEF queens'(Q ⊎ {q}, c) =
    ∪ / (f * Pos(q, _))
    WHERE
    f(pos) = IF c(pos) THEN queens'(Q, friendly(pos) ∧ c) ⊗ {q ↦ pos}
                      ELSE ∅                                          FI
```

Unsere neue Implementierung hat gegebenüber der aus Programm 16.6 zwei Vorteile: Eine Dame wird überhaupt nur dann auf dem Schachbrett platziert, wenn ihre Position mit der der nachfolgend zu platzierenden kompatibel ist. Und diese Prüfung können wir – indem wir sie als Bedingungsteil eines IF-Konstrukts ausdrücken – sehr zeitig, schon während des Aufbaus der Constraint-Konjunktion durchführen, so dass ungültige Konfigurationen gar nicht erst erzeugt werden. Das Programm 16.7 hat aber gleichzeitig den Nachteil, dass wir keine *invariant code motion* wie in Programm 16.6 mehr durchführen können und so wieder an Effizienz einbüßen.

16.3.2 Ein allgemeines Schema für die globale Suche

Unser Ziel ist jetzt, ein allgemeines Schema bzw. eine generische Funktion für die globale Suche anzugeben. Das können wir durch Abstraktion des n-Damen-Beispiels einfach erreichen.

Wir stellen einen Suchraum als Menge von Abbildungen vom Definitionsbereich A in den Wertebereich B dar. Zur Suchraumreduktion verwenden wir die Prädikate $\boxdot_{ran}$, $\boxdot$ und $\boxtimes$ aus Programm 15.6.

Programm 16.8 zeigt eine Spezifikation der allgemeinen Suchfunktion *globalSearch*. Sie bekommt als Argumente den Prädikaten entsprechende Constraints p_r, p und m sowie den Definitionsbereich A und den Wertebereich B und filtert den von A und B aufgespannten Suchraum bzgl. der Constraints.

Da wir wie bisher von totalen Maps zur Bescheibung von Lösungen ausgehen, ist eine Filterung von Suchräumen mit einem Prädikat p_d auf dem Definitionsbereich (also $\boxdot_{dom}\ p_d$) nicht sinnvoll: Entweder bliebe der Suchraum unverändert, weil das Prädikat p_d für alle Elemente des Definitionsbereichs gilt. Oder es gibt Elemente in A, für die p_d nicht gilt. Da wir totale Maps betrachten, gibt es diese dann in jeder Map des Suchraums und somit wäre der reduzierte Suchraum leer.

Programm 16.8 Die generische Suchfunktion *globalSearch*

FUN $globalSearch$:
$(\beta \to Bool) \times (\alpha \times \beta \to Bool) \times ((\alpha \times \beta) \to (\alpha \times \beta) \to Bool) \to$
$Set\ \alpha \times Set\ \beta \to Set(Map\ \alpha\ \beta)$

PROP $globalSearch(p_r, p, m)(A, B) = [\![A \rightarrowtail B]\!] \triangleright (\boxdot_{ran}\ p_r) \triangleright (\boxdot\ p) \triangleright (\boxtimes\ m)$

DEF $globalSearch(p_r, p, m)(A, B) = search(p, m)(A,\ B \triangleright p_r)$

FUN $search: (\alpha \times \beta \to Bool) \times ((\alpha \times \beta) \to (\alpha \times \beta) \to Bool) \to$
$Set\ \alpha \times Set\ \beta \to Set(Map\ \alpha\ \beta)$

PROP $search(p, m)(A, B) = [\![A \rightarrowtail B]\!] \triangleright (\boxdot\ p) \triangleright (\boxtimes\ m)$

Mit Theorem *B2* schränken wir zunächst den Wertebereich ein und kommen so zu einer Funktion *search*, deren Form der der *queens*-Funktion aus Abschnitt 16.3.1 entspricht. Die Funktion *search* können wir jetzt nahezu analog zu *queens* aus dem vorherigen Abschnitt entwickeln, denn sie ist einfach eine Verallgemeinerung von *queens*. Abbildung 16.7 zeigt die Ableitung einer rekursiven Definition. Das Ergebnis der Transformation ist in Programm 16.9 angegeben.

Programm 16.9 Effiziente Suchraumeinschränkung mit $\boxdot$ und $\boxtimes$

DEF $search(p, m)(\emptyset, \_) = \{\square\}$

DEF $search(p, m)(A \uplus \{a\},\ B) =$
LET $S = search(p, m)(A, B)$
$f(b) = (S \triangleright (\boxdot\ m(a, b))) \otimes \{a \mapsto b\}$
IN $\cup\ /\ f * (B \triangleright p(a, \_))$

$$
\begin{aligned}
& search(p,m)(\emptyset, \_) = \{\Box\} \\
& search(p,m)(A \uplus \{a\}, B) = [\![\, A \uplus \{a\} \rightarrowtail B \,]\!] \triangleright (\boxdot\, p) \triangleright (\boxtimes\, m) \\
& \overset{A1}{\equiv} ([\![\, A \rightarrowtail B \,]\!] \otimes [\![\, \{a\} \rightarrowtail B \,]\!]) \triangleright (\boxdot\, p) \triangleright (\boxtimes\, m) \\
& \overset{B4}{\equiv} (([\![\, A \rightarrowtail B \,]\!] \triangleright (\boxdot\, p)) \otimes ([\![\, \{a\} \rightarrowtail B \,]\!] \triangleright (\boxdot\, p))) \triangleright (\boxtimes\, m) \\
& \overset{B3}{\equiv} (([\![\, A \rightarrowtail B \,]\!] \triangleright (\boxdot\, p)) \otimes [\![\, \{a\} \rightarrowtail B \triangleright p(a, \_) \,]\!]) \triangleright (\boxtimes\, m) \\
& \overset{B5}{\equiv} (([\![\, A \rightarrowtail B \,]\!] \triangleright (\boxdot\, p \wedge \boxtimes\, m)) \otimes [\![\, \{a\} \rightarrowtail B \triangleright p(a, \_) \,]\!]) \triangleright (\boxtimes\, m) \\
& \overset{C2}{\equiv} (([\![\, A \rightarrowtail B \,]\!] \triangleright (\boxdot\, p) \triangleright (\boxtimes\, m)) \otimes [\![\, \{a\} \rightarrowtail B \triangleright p(a, \_) \,]\!]) \triangleright (\boxtimes\, m) \\
& \overset{Ind}{\equiv} (search(p,m)(A,B) \otimes [\![\, \{a\} \rightarrowtail B \triangleright p(a, \_) \,]\!]) \triangleright (\boxtimes\, m) \\
& \overset{A2}{\equiv} (search(p,m)(A,B) \otimes \bigcup_{b \in B \triangleright p(a, \_)} \{a \mapsto b\}) \triangleright (\boxtimes\, m) \\
& \overset{A3}{\equiv} (\bigcup_{b \in B \triangleright p(a, \_)} search(p,m)(A,B) \otimes \{a \mapsto b\}) \triangleright (\boxtimes\, m) \\
& \overset{C1}{\equiv} \bigcup_{b \in B \triangleright p(a, \_)} ((search(p,m)(A,B) \otimes \{a \mapsto b\}) \triangleright (\boxtimes\, m)) \\
& \overset{B6}{\equiv} \bigcup_{b \in B \triangleright p(a, \_)} (((search(p,m)(A,B) \triangleright (\boxtimes\, m)) \triangleright (\boxdot\, m(a,b))) \otimes \{a \mapsto b\}) \\
& \overset{Def}{\equiv} \bigcup_{b \in B \triangleright p(a, \_)} ((search(p,m)(A,B) \triangleright (\boxdot\, m(a,b))) \otimes \{a \mapsto b\})
\end{aligned}
$$

Abb. 16.7: Ableitung des allgemeinen Suchschemas

Ganz analog können wir schließlich zu einer Akkumulation von Constraints für eine effiziente Implementierung übergehen. Wir lassen die Ableitung, die völlig synchron zu der in Abbildung 16.6 verläuft, aus und geben stattdessen nur das Ergebnis in Programm 16.10 an.

Programm 16.10 Suchraumeinschränkung mit akkumulierten Prädikaten

```
DEF search(p,m)(A,B) = search'(p,m)(A,B,λ_, _ • true)
FUN search': (α × β → Bool) × ((α × β) → (α × β) → Bool) →
             Set α × Set β × (α × β → Bool) → Set(Map α β)
DEF search'(p,m)(∅, _, c) = {□}
DEF search'(p,m)(A ⊎ {a}, B, c) =
    ∪ / f * (B ▷ p(a, _))
    WHERE
    f(b) = IF c(a,b) THEN search'(p,m)(A, B, m(a,b) ∧ c) ⊗ {a ↦ b}
                     ELSE ∅                                  FI
```

In Programm 16.11 ist eine Lösung des n-Damen-Problems auf der Basis der Funktion *globalSearch* angegeben. Während wir in der Spezifikation von *queens* in Programm 16.2 und bei der Programmableitung in Abschnitt 16.3.1

eine Prüfung $\boxtimes_{ran}$ des Prädikats *friendly*: $Position \times Position \to Bool$ vorgenommen haben, sind wir jetzt von dem allgemeineren Prädikat $\boxtimes$ ausgegangen und überladen hier daher *friendly* mit einer entsprechenden Version.

Programm 16.11 Lösung des n-Damen-Problems mit *globalSearch*

FUN $queens: Set\ Queen \to Set\ Map(Queen, Position)$
DEF $queens(Q) = globalSearch(\lambda\_ \bullet true, onRow, friendly)(Q, Pos(Q))$
FUN $friendly(\_, p_1)(\_, p_2) = friendly(p_1)(p_2)$

Partielle Maps

Mit Hilfe von Maps haben wir Lösungen von Suchproblemen in Form von Zuodnungen von Werten zu Variablen dargestellt. Eine Lösung ordnete dabei *jeder* Variablen einen Wert zu. Daher haben wir mit Mengen totaler Maps gearbeitet.

Möchte man die Ideen und Vorgehensweise der vorhergehenden Abschnitte auf partielle Maps übertragen, dann muss man sich darüber im Klaren sein, dass man nun eine andere Art von Problemen betrachtet. So enthält der Suchraum $[\![A \to B]\!]$ im Gegensatz zur totalen Version $[\![A \rightarrowtail B]\!]$ nicht nur alle Maps mit Definitionsbereich A sondern auch jede Map, deren Definitionsbereich eine Teilmenge von A ist. Man sucht also erstens in einem im Allgemeinen sehr viel größeren Suchraum und lässt zweitens auch partielle Zuordnungen zu.

Dadurch ist es nun auch sinnvoll, einen Suchraum mit $\boxdot_{dom}$ zu filtern. Das kann man per Theorem *B1* einfach auf eine Filterung des Definitionsbereichs übertragen und kommt so zu einer zur totalen Version vergleichbaren Spezifikation. Von hier an ist dann das Vorgehen bei der Programmableitung weitgehend analog.

16.4 Beispiel: Scheduling

Unser allgemeines Suchschema ist auf vier Arten von Prädikaten beschränkt, nämlich „punktweise“ („pointwise“) Bedingungen auf den Elementen des Definitions-, des Wertebereichs und den Mapelementen, sowie Bedingungen auf je einem Paar von Mapelementen („mutually“). Damit kann man Probleme mit lokalen Constraints auf einzelnen Objekten bzw. zwischen je zwei Objekten beschreiben. Die Constraints müssen dabei homogen sein, also gleichmäßig für alle Objekte des Problems und somit für alle Punkte der Map gelten.

Trotz dieser Einschränkungen kann man mit unserem allgemeinen Suchschema schon eine ganze Reihe von Problemen bequem beschreiben und effizient lösen.

Das globale Constraint *alldifferent* [78], das einer Menge von Variablen Werte zuweist, so dass allen Variablen unterschiedliche Werte zugeordnet werden, kann man z. B. in einfacher Weise auf eine Menge lokaler Constraints abbilden. Das zeigt Programm 16.12.

Programm 16.12 Das Constraint *alldifferent* als globale Suche

```
FUN alldifferent: Set Var × Set Value → Set(Map Var Value)
DEF alldifferent (vars, values) =
    globalSearch((λ _ • true), (λ _, _ • true), (λ x1, y1 • λ x2, y2 • y1 ≠ y2))
                 (vars, values)
```

Hat man anders gelagerte Probleme, die z. B. Constraints auf jeweils mehreren Punkten der Map notwendig machen, dann muss man entweder versuchen, durch geeignete Wahl von Definitions- und Wertebereich das Problem an unsere Form anzupassen, oder man berechnet eine Teillösung und schränkt diese im Nachhinein ein. Eine weitere Möglichkeit besteht darin, das Suchschema entsprechend den in den vorherigen Abschnitten gezeigten Verfahren für neue Anforderungen anzupassen.

Wir wollen im Folgenden noch eine weitere Anwendung unseres Suchschemas auf ein eingeschränktes Scheduling-Problem skizzieren. Unser Beispiel geht auf Pepper und Smith [113] zurück, die den Anwendungsbereich Scheduling ausführlich behandeln und zeigen, wie man durch geschickte Transformationen auch mit komplizierteren Scheduling-Problemen umgehen kann.

Wir betrachten ein Transport-Planungsproblem. Eine Lösung haben wir in Programm 16.13 angegeben. Eine Menge von Frachteinheiten vom Typ *Cargo* soll von einem Lager abgeholt und zu einer Verkaufsstelle oder einem Markt transportiert werden. Jede Frachteinheit hat einen bestimmten Zeitbereich, innerhalb dessen sie vom Lager abgeholt sein muss. Start- und Endzeit *start* und *final* dieses Zeitbereichs seien hierbei durch den Zeitpunkt der Anlieferung der Fracht im Lager, Verfallsdatum der Waren, Lagerkapazität etc. bestimmt. Jede Frachteinheit hat außerdem eine bestimmte Größe *size* und der Transporter hat eine begrenzte Kapazität, so dass er pro Fahrt nur eine beschränkte Menge von Waren transportieren kann.

Unser Transportproblem ist stark vereinfacht. Größere Scheduling-Probleme umfassen normalerweise ganze Flotten von Transportfahrzeugen, Schiffen oder Flugzeugen und ebenso unterschiedliche Transportziele, -wege und -zeiten. In [113] wird der Umgang mit einem solchen erweiterten Problem gezeigt. Bei echten Anwendungen spielt häufig zusätzlich auch noch eine Schichtplanung für das Personal eine Rolle.

Programm 16.13 Ein eingeschränktes Planungsproblem

```
TYPE Schedule = Map Cargo Trip
TYPE Cargo = cargo(id: String, start: Int, final: Int, size: Int)
TYPE Trip = trip(start: Int)

FUN fit: Cargo × Trip → Bool
DEF fit(cargo, trip) = start(cargo) ≤ start(trip) ∧ start(trip) ≤ final(cargo)

FUN sep: Cargo × Trip → Cargo × Trip → Bool
DEF sep(_, trip1)(_, trip2) =
    (start(trip1) = start(trip2)) ∨ ( |start(trip1) − start(trip2)| ≥ time)

FUN schedules: Set Cargo × Set Trip → Set Schedule
DEF schedules(cargos, trips) = globalSearch(λ _ • true, fit, sep)(cargos, trips)
```

Wir wollen für unser Problem die zulässigen Transportpläne berechnen, so dass jeweils alle Waren innerhalb ihrer Start- und Endzeiten vom Lager zum Markt transportiert werden. Einen Plan vom Typ *Schedule* repräsentieren wir durch eine Abbildung aus der Menge der Frachteinheiten in die Menge der Transporte. Da bei uns alle Transporte die gleiche Zeit brauchen, ist ein Transport vom Typ *Trip* lediglich durch seinen Startzeitpunkt *start* gekennzeichnet.

Ausgehend von der Menge der Frachteinheiten, diese repräsentiert den Definitionsbereich, und der Menge der möglichen Transporte, dem Wertebereich, berechnet die Funktion *schedules* eine Menge von Transportplänen. Dazu verwenden wir die allgemeine Suchfunktion *globalSearch* mit den Scheduling-Constraints *fit* und *sep*.

Das Prädikat *fit* prüft, ob die Startzeit eines Transports innerhalb des Zeitbereichs der Frachteinheit liegt, in dem diese vom Lager abgeholt werden muss. Dieses Prädikat wird „pointwise“, d. h. auf jeden Punkt einer Map angewendet, die einen Transportplan beschreibt.

Wenn wir annehmen, dass ein Transport immer genau *time* Zeiteinheiten benötigt, dann können wir mit dem Prädikat *sep* festlegen, dass die Startzeiten von je zwei Transporten mindestens um diese *time* Zeiteinheiten differieren müssen. Das Prädikat *sep* muss auf je zwei Punkten der Transportmap („mutually“) gelten.

Die Funktion *schedules* berechnet nun Transportpläne, die allerdings noch nicht die Kapazität *capacity* des Transporters und die Größe der jeweiligen Fracht berücksichtigen. Ein solches Constraint enthält Abhängigkeiten von Mengen von Frachteinheiten und keine unserer o. g. vier Prädikatarten wäre hierfür ausreichend.

Man kann ein solches Kapazitäts-Constraint aber als ein („pointwise“) Prädikat angeben, wenn man eine andere Problem-Repräsentation wählt und statt einzelner Frachteinheiten Mengen von Frachteinheiten auf Transporte abbildet:

FUN $cap\colon Set\ Cargo \times Trip \to Bool$
DEF $cap(cargos, \_) = (+ / ((\lambda cargo\,.\,size(cargo)) * cargos)) \leq capacity$

Dann müssten aber auch alle anderen Constraints, der Datentyp *Schedule* und die Funktion *schedules* auf den neuen Definitionsbereich geliftet werden.

Hier zeigt sich die o. g. Einschränkung unserer allgemeinen Suchfunktion *globalSearch*: Eine Definition eines Kapazitäts-Constraints durch Anpassung des Definitionsbereichs ist zwar prinzipiell wie eben beschrieben möglich. Da wir im gezeigten Verfahren aber davon ausgehen, dass initial sämtliche Elemente von Definitions- und Wertebereich zur Verfügung stehen, müssen wir nun einen erheblichen Mehraufwand bei der Generierung der Elemente des Definitionsbereichs in Kauf nehmen.

17

Zeit und Zustand in der funktionalen Welt

The future will be better tomorrow.
George W. Bush
Die Zeit ist auch nicht mehr das, was sie mal war. *Albert Einstein*

Zeit ist ein in jeder Hinsicht spannendes Phänomen; man kann ihre Geschichte sowohl kurz als auch kürzest erzählen [70, 71]. Bei Schiller enteilet sie und bei Chamisso naht sie, doch bei beiden tut sie's unaufhaltsam. Für Gottfried Keller dagegen steht sie still und wir ziehen durch sie hindurch. Für Joubert schließlich ist sie alleine Bewegung im Raum und für Bloch gibt es sie nur, wo etwas geschieht.

In der Funktionalen Programmierung möchte man die Zeit am liebsten ganz los werden, denn Funktionen sind im wahrsten Sinn des Wortes Zeit-los: Der Wert von $\sin(\frac{\pi}{5})$ sollte zu Weihnachten kein anderer sein als zu Ostern. Aber spätestens, wenn Interaktionen mit der Umwelt nötig werden – sei es mit Benutzern am Bildschirm oder mit Sensoren und Aktuatoren in einer Prozesssteuerung –, kann man die Zeit nicht mehr ignorieren. Denn die Welt lebt in der Zeit.

Damit müssen, ob sie es wollen oder nicht, auch die funktionalen Programmierer irgendwie mit dem Phänomen Zeit umgehen. Dazu gibt es eine Reihe von Vorschlägen, aber so ganz überzeugen kann keiner davon. Mit anderen Worten: Die Frage ist noch immer ein offenes Forschungsthema. Wir werden in diesem Kapitel den im Augenblick am weitesten akzeptierten Lösungsansatz diskutieren (nämliche Monaden) und in den folgenden Kapiteln darauf aufbauende Erweiterungen präsentieren.

Wir werden allerdings versuchen, das Thema etwas grundsätzlicher anzugehen, so dass sich die aktuell favorisierten konkreten Sprach- und Programmierkonzepte besser in die generelle wissenschaftliche Fragestellung einordnen lassen und vielleicht sogar Optionen für bessere Lösungsansätze sichtbar werden.

17.1 Zeit und Zustand: Zwei Seiten einer Medaille

Aller Zustand ist gut, der natürlich ist und vernünftig.
Goethe (Hermann und Dorothea)

In der Programmierung wird über *Zeit* eigentlich selten gesprochen;[1] meistens geht es um den Begriff *Zustand*. Aber diese beiden Konzepte sind untrennbar miteinander verbunden: Bei einem System lohnt es sich nur, über Zeit zu reden, wenn das System sich mit dem Lauf der Zeit ändert. Und das, was sich da ändern kann, wird üblicherweise unter dem Begriff „Zustand des Systems" subsumiert.

Was genau diese „Zustände" sind, ist im Allgemeinen nicht einmal vollständig klar. Wenn Einstein sagt „Felder sind physikalische Zustände des Raumes", dann hat er damit weder Felder noch den Raum vollständig charakterisiert, aber es ist doch ein erster Schritt zum besseren Verständnis beider Dinge getan. Das ist sogar typisch für zustandsorientierte Systembeschreibungen: Die Zustände werden meistens nicht vollständig angegeben (oft ist das sogar unmöglich); man beschränkt sich stattdessen auf die Beschreibung ausgewählter Aspekte. (In der Mathematik führt das auf den Übergang von einer algebraischen zu einer coalgebraischen Sichtweise.)

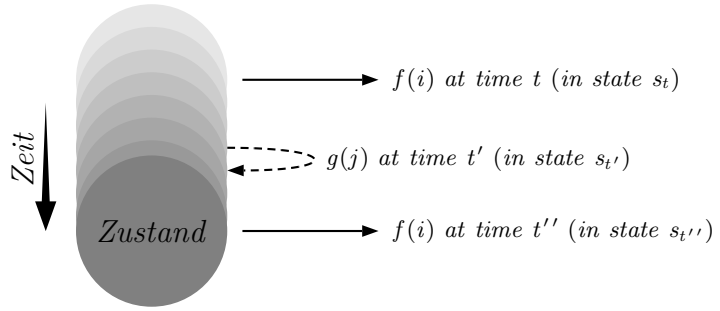


Abb. 17.1: Systemzustände und Programm

In Abbildung 17.1 wird das Zusammenspiel von Zustand und Zeit aus Sicht eines Programms illustriert: Ein System entwickelt sich im Lauf der Zeit durch eine Folge von Zuständen (hier als diskrete Folge skizziert, tatsächlich aber meistens als kontinuierlicher Prozess). Aus Sicht des Programms gibt es zwei relevante Aspekte:

- Unser Programm greift immer wieder über eine „Funktion" f auf den Systemzustand zu, z. B. indem es den Wert des i-ten Sensors ausliest.

[1] Eine Ausnahme bilden die so genannten Realzeit-Applikationen; doch dabei geht es eigentlich mehr um „Uhren" als um Zeit.

Das führt aus Sicht der Funktionalen Programmierung auf ein Problem: f ist im mathematischen Sinn keine Funktion, weil abhängig vom Zeitpunkt der Aufruf $f(i)$ unterschiedliche Werte liefert. Wir verwenden daher im Folgenden lieber den allgemeineren Begriff *Operation* als den spezielleren Begriff Funktion.
Um echte Funktionen zu erhalten, müsste man den Zeitpunkt als weiteres Argument hinzufügen, so dass in Abbildung 17.1 stehen würde $f(i)(t)$ und $f(i)(t'')$. Aber das führt auf eine Reihe weiterer Komplikationen, die wir gleich noch diskutieren werden.

- Zuvor müssen wir auf eine weitere Schwierigkeit eingehen, die sogar noch unangenehmer ist als das Problem mit der Operation $f(i)$. Solange das System sich einfach nur (selbstständig) ändert und wir es mittels Operationen wie $f(i)$ *beobachten* wollen, kann man das noch relativ leicht in das funktionale Paradigma einbauen.
  Schlimmer wird es jedoch, wenn wir den Systemzustand aus dem Programm heraus *aktiv ändern* wollen, z. B. indem wir auf den j-ten Aktuator ein Signal geben. Das wird in Abbildung 17.1 durch die „Funktion“ $g(j)$ angedeutet. Ein solches $g(j)$ hat im Allgemeinen überhaupt keinen Wert mehr, sondern nur noch einen *Effekt*. Aber Dinge, die keinen Wert darstellen, liegen völlig quer zu den Prinzipien funktionaler Programmierung. Wir sprechen deshalb lieber von *Aktionen* als von Funktionen.

Schließlich gibt es aus pragmatischer Sicht auch noch den Wunsch, beide Varianten zu verschmelzen, also Aktionen zu haben, die sowohl den Zustand ändern als auch ein Resultat liefern. Ein typisches Beispiel ist die Operation *readNext*(*file*), die als Ergebnis das nächste Element aus der angegebenen Datei liefert und als Effekt den Lesezeiger in der Datei weiterschiebt.

Definition (Operation, Beobachtung, Aktion)

Insgesamt müssen wir uns mit vier Varianten von „Funktionen“ herumplagen:

1. Echte *Funktionen* im mathematischen Sinn, deren Wert jeweils nur von den angegebenen Argumenten abhängt und nicht vom Zeitpunkt des Aufrufs.
2. Reine *Beobachtungen*, deren jeweiliges Resultat nicht nur von den Argumenten, sondern auch vom Zeitpunkt des Aufrufs abhängt.
3. Reine *Aktionen*, die nur einen Effekt haben (also eine Zustandsänderung bewirken), aber kein Resultat liefern.
4. Allgemeine *Operationen mit Seiteneffekt* (also *Aktionen mit Ergebnis*), die sowohl einen Effekt auslösen als auch ein – zeitabhängiges – Resultat liefern.

Als Oberbegriff für alle diese Varianten verwenden wir das Wort ***Operation***.

17.1.1 Ein kleines Beispiel

Um die Diskussion nicht allzu abstrakt zu führen, betrachten wir ein kleines Beispiel. Dieses Beispiel hat den Charme, so weit abgespeckt zu sein, dass man es leicht und knapp diskutieren kann. Aber es ist trotzdem repräsentativ für eine Fülle von praktisch relevanten Aufgaben, z. B. im Compilerbau, wo abstrakte Syntaxbäume simultan mit Symboltabellen und Fehlermeldungen behandelt werden müssen, oder in Spielen, in denen baumartige Spielstrategien mit Benutzerinteraktionen gemischt werden.

Abbildung 17.2 illustriert die Situation in UML-artiger Notation: Wir haben einen Evaluator, der arithmetische Ausdrücke (die bereits in der Form von abstrakten Syntaxbäumen vorliegen) auswerten kann. Die Ausdrücke können Variablen enthalten, deren aktuelle Werte jeweils vom Benutzer interaktiv zu erfragen sind. Damit aber eine Variable nicht mehrfach angefordert wird, sollen die Eingaben in einem Cache gespeichert werden.

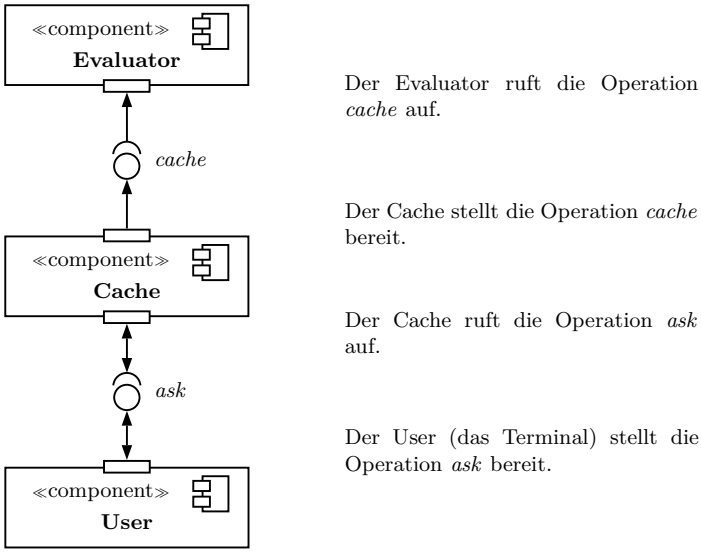


Abb. 17.2: Ein Evaluator mit Cache und Benutzereingabe

Im Folgenden wollen wir versuchen, uns dieser Programmieraufgabe unter einer funktionalen Sichtweise Stück für Stück zu nähern. Dabei nehmen wir zunächst eine „naiv-optimistische“ Sichtweise ein und schreiben die Dinge so knapp und elegant hin, wie wir es am liebsten hätten. Dann werden wir diskutieren, warum es ganz so schlicht und idealistisch nicht geht.

Wir beginnen mit den elementaren und unproblematischen Grundlagen. Die Syntaxbäume werden durch einen klassischen Baumtyp definiert, der für

jede Art von arithmetischer Operation eine Variante besitzt. An den Blättern können sowohl Konstanten als auch Variablen vorkommen:

```
TYPE Tree = add( left: Tree, right: Tree )   -- Addition
          | sub( left: Tree, right: Tree )   -- Subtraktiuon
          | ...
          | const( value: Int )              -- Konstante
          | var( name: String )              -- Variable
```

Die Auswertung eines solchen Baums würde man gern in klassischer musterbasierter Form schreiben, angelehnt an die Baumstruktur:

```
FUN eval: Tree → Int
DEF eval(add(l, r)) = eval(l) + eval(r)
DEF eval(sub(l, r)) = eval(l) − eval(r)
    ...
DEF eval(const(n)) = n
DEF eval(var(x)) = cache(x)          -- idealistisches Wunschdenken
```

Das Problem ist offensichtlich die letzte Zeile: Die Auswertung der Variablen verlangt eine Interaktion mit dem Benutzer – und das ist funktional nicht so einfach möglich. Der obige Versuch stellt daher nur Wunschdenken dar.

Bevor wir weiter in die technischen und konzeptuellen Details einsteigen, wollen wir den Rahmen festlegen, in dem das geschieht. Programm 17.1 definiert ein entsprechendes „Package“ mit drei „Komponenten“, wobei wir erst einmal offen lassen müssen, was solche „Komponenten“ eigentlich sind:

Programm 17.1 Interaktiver Evaluator mit Cache

```
PACKAGE System = {
  COMPONENT Evaluator = {
    USE Cache
    TYPE Tree = ...
    FUN eval: Tree → Int
    ...
    DEF eval(var(x)) = cache(x)          -- idealistischer Versuch
  }
  COMPONENT Cache = {
    FUN cache: String → Int MEMOIZED     -- idealistischer Versuch
    DEF cache(x) = User.ask(x)           -- idealistischer Versuch (Typfehler!)
  }
  COMPONENT User = {
    FUN ask: String → IO Int             -- monadisch
    DEF ask(x) = ? ? ?                   -- Interaktion mit dem Betriebssystem
  }
}
```

Das fundamentale Problem wird hier anhand eines *Typfehlers* deutlich sichtbar. Die Operation $ask(x)$ hat ein Ergebnis vom (monadischen) Typ $IO\ Int$, während $cache(x)$ einen Wert vom Typ Int verlangt. Und unser Trick, dieses fundamentale Problem über *Memoization* abzufangen, ist, gelinde gesagt, etwas blauäugig.

Wenn wir diese Memoization nicht im Compiler verstecken, sondern explizit ausprogrammieren wollten, dann müsste die Komponente *Cache* etwa folgendes Aussehen haben:

```
COMPONENT Cache(map: Map String Int) = {        -- ----------
  DEF cache(x) = IF x ∈ map THEN map(x)         -- Ad-hoc-Notation
                 IF x ∉ map THEN k              -- ----------
                                EVOLVE Cache(map')
                                WHERE k ← User.ask(x)
                                      map' = (map x ⇐ k)
}
```

Diese Form löst zwar noch immer nicht das Typproblem, zeigt aber zumindest das elementare konzeptuelle Problem: *Das Beschaffen des nächsten Wertes vom Benutzer bewirkt eine Änderung des Caches.* Mit anderen Worten: Der Cache erfährt eine Evolution – und das ist genau das Charakteristikum der Zustands-Monaden aus Abschnitt 11.2.3.

Diese Beobachtung hat dazu geführt, dass die Zustands-Monaden zur Standardtechnik avancierten, mit der in funktionalen Sprachen das Problem der Interaktion mit der Umwelt – kurz: Ein-/Ausgabe – gelöst werden soll. Im Folgenden präsentieren wir deshalb ganz kurz die gängigen Ansätze zur Monaden-basierten Ein-/Ausgabe. Danach wenden wir uns wieder dem fundamentalen Problem ihrer Integration in die „echte“ Funktionale Programmierung zu.

17.2 *Monaden*: Ein schicker Name für Altbekanntes

Nach dem heutigen Stand der Kunst ist die Verwendung von Zustands-Monaden das gängige Mittel, um das Ein-/Ausgabe-Problem für funktionale Sprachen in den Griff zu bekommen [142]. Allerdings handelt es sich bei diesem Lösungsansatz nicht um eine ganz neue, geniale Erfindung, sondern eher um das Zusammenbringen von Erkenntnissen und Entwicklungen aus mehreren Bereichen[2] [75]. Diese Entstehungsgeschichte ist in Abbildung 17.3 skizziert.

Die eigentliche Geburtsstunde dieser Ideen liegt in der Entwicklung der so genannten *Denotationellen Semantik*, die vor allem durch Dana Scott und Christopher Strachey Anfang der 70er Jahre initiiert wurde. Bei dieser Form der formalen Semantikdefinition werden Programme als Funktionen charakterisiert. Um das auch mit imperativen Programmen – einschließlich des heute

[2] Schon 1965 hat Peter Landin den Zusammenhang in dieser Form beschrieben, als er eine Beziehung zwischen ALGOL 60 und dem λ-Kalkül diskutierte [90].

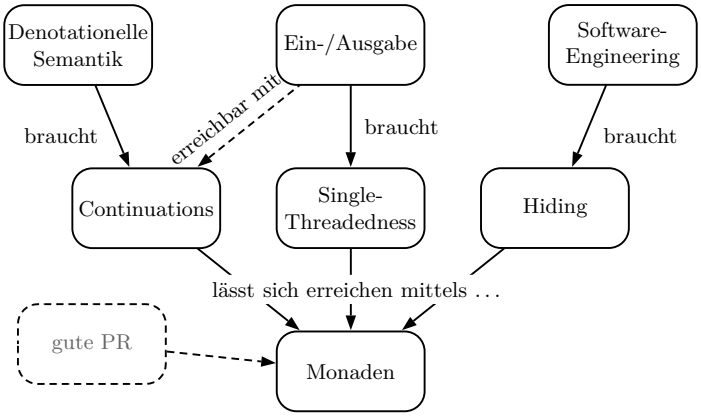


Abb. 17.3: Genealogie der Monadenverwendung

verpönten `goto`-Statements – tun zu können, braucht man so genannte *Continuations* (s. Abschnitt 17.2.1 weiter unten).

Man erkannte schnell, dass mit diesen Continuations das Ein-/Ausgabe-Problem funktionaler Programme lösbar war. Allerdings gibt es eine Schwierigkeit: Man kann Dinge programmieren, die im Widerspruch zu den physikalischen Gegebenheiten der realen Welt stehen. Was man zusätzlich noch braucht ist die Garantie der *Single-Threadedness* (vgl. Abschnitt 13.1). Wie wir schon in Kapitel 13 gesehen haben, ist diese Eigenschaft wesentlich für die Effizienz vieler Algorithmen, die mit großen Datenstrukturen arbeiten. Im Zusammenhang mit der Ein-/Ausgabe wird sie noch bedeutender: Hier ist Single-Threadness Voraussetzung, um Widersprüche zur Realität der physikalischen Welt zu vermeiden.

Ein dritter Aspekt kam aus dem Software-Engineering hinzu: Dort hatte sich die Erkenntnis durchgesetzt, dass Modularisierung mit dem so genannten *Hiding*-Prinzip ein essenzielles Kriterium guter Programmierung ist.

Wenn man alle drei Konzepte – Continuations, Single-Threadedness und Hiding – zusammennimmt, entsteht eine Technik, mit der sich das Ein-/Ausgabe-Problem der Funktionalen Programmierung relativ gut und angemessen behandeln lässt.

So richtig griffig wurde das Ganze aber erst durch einen Glücksfall: Es gab im Rahmen der Denotationellen Semantik eine Beobachtung, die im Wesentlichen auf Arbeiten von Moggi [99] beruhte: Die Continuation-Techniken zusammen mit der Single-Threadedness führen auf eine Sammlung von Funktionen, die „zufällig“ gerade die Axiome der *Monaden* aus der Kategorientheorie erfüllen (vgl. Kapitel 11). Und besonders gut traf es sich, dass diese Monaden auch noch das Hiding-Prinzip respektieren.

Auch wenn man für die Programmierung von Ein-/Ausgabe nichts von Moggis tiefschürfender Mathematik benötigte, so lieferte seine Erkenntnis

doch etwas, das man sehr gut gebrauchen konnte: einen schicken Namen, der sich trefflich für PR eignete.

Damit hatte sich die Programmiertechnik der monadischen Ein-/Ausgabe etabliert (in OPAL noch unter dem Namen Commands, in HASKELL bereits unter dem Namen Monaden). Und seither hat es eine intensive Analyse und Weiterentwicklung der Anwendungsmöglichkeiten von Monaden in der Programmierung gegeben. In den folgenden Abschnitten werden wir – allerdings nur kurz und exemplarisch – auf einige dieser Aspekte eingehen.

Anmerkung 1: Interessanterweise haben viele Ansätze, darunter auch OPAL *und* HASKELL*, ursprünglich versucht, das Ein-/Ausgabe-Problem mit einer anderen Technik zu lösen, nämlich mit so genannten Strömen (engl.: Streams). Diese Ströme sind nichts anderes als lazy Listen (vgl. Kapitel 2). Auf den ersten Blick erschienen Ströme als das ideale Bindeglied zwischen funktionaler und zeitbehafteter Welt, aber in der praktischen Arbeit erwies sich schnell, dass die Programme – vor allem wenn es um dialogartige Situationen ging – völlig gegen die Intuition verstießen und damit ziemlich mystisch und unverständlich wurden. Deshalb wurde z. B. in* OPAL *schon Mitte der 80er Jahre über den Strömen der Typ Com*[α] *implementiert, der genau das realisierte, was heute Zustands-Monaden heißt. Die Erkenntnis, dass eine direkte Implementierung ohne den Umweg über die Ströme viel effizienter funktioniert, hat dann zur endgültigen Eliminierung der Ströme aus dem* OPAL*-System geführt. In* HASKELL *kam man unabhängig davon kurze Zeit später zur gleichen Erkenntnis, so dass auch dort die Ströme durch Monaden ersetzt wurden.*

Ströme werden auch heute noch in einigen Ansätzen im Software-Engineering benutzt, aber auch hier zeigt sich, dass sie eigentlich nur dort brauchbar sind, wo so genannte Datenfluss-Architekturen betrachtet werden. Bei allen Arten von Interaktions-orientierten Architekturen – wie z. B. Client-Server-Systemen – stößt man auf die gleichen Probleme wie bei der Funktionalen Programmierung.

Anmerkung 2: Man hätte die Single-Threadedness auch mit Hilfe von linearen Typen (vgl. Abschnitt 13.1.3) erreichen können (was in CLEAN *im Wesentlichen gemacht wird [119]). Aber diese erfordern eine relativ komplexe Typanalyse. Bei der Verwendung von Monaden wird der gleiche Zweck erreicht, aber es genügt das klassische Hindley-Milner-Typsystem.*

17.2.1 Programmieren mit *Continuations*

Die Idee der *Continuations* ist eigentlich ganz einfach, wie wir schon in Abschnitt 1.4.2 gesehen haben. Man gibt einer Funktion ihre „Fortsetzung“ als Argument mit. Seien z. B. die folgenden Funktionen gegeben:

FUN $f\colon\ \alpha \to \beta$
FUN $g\colon\ \beta \to \gamma$
$\quad\ldots$
DEF $h(\ldots) = \ldots g(f(a))\ldots$

Hier kann man f so in eine Funktion f' umdefinieren, dass es seine Fortsetzung g als zusätzliches Argument mitbekommt. Dementsprechend ändert sich die Applikation in h:

FUN $f'\colon\ \alpha \to (\beta \to \gamma) \to \gamma$
DEF $f'(x)(\varphi) = \varphi(f(x))$
$\ldots$
DEF $h(\ldots) = \ldots f'(a)(g) \ldots$

Dies ist ganz offensichtlich ein kleiner, harmloser Programmiertrick, der für sich genommen noch wenig Effekt hat. Aber wenn man ihn systematisch einsetzt, kann man einige recht nette Dinge damit tun. Das kann man z. B. in der ursprünglichen Arbeit von Mitchell Wand [143] sehr schön nachlesen, aber auch in einer Reihe von späteren Arbeiten, vor allem im Umfeld der Sprachen SCHEME und LISP (s. [3]).

Einige Anwendungen – nützliche ebenso wie skurrile – haben wir bereits in Abschnitt 1.4.2 gesehen. Aber neben ihrer Rolle in cleveren Programmiertricks bieten Continuations auch eine Möglichkeit, die Ein-/Ausgabe in funktionalen Programmen sauber zu definieren – sofern man diszipliniert mit ihnen umgeht. Betrachten wir ein Minibeispiel. Seien die beiden folgenden Funktionen zum Lesen und Schreiben gegeben:

FUN *read*: *State* $\to \alpha$
FUN *write*: *String* $\to$ *State* $\to$ *State*

Dabei stehe *State* für den „Maschinenzustand“. Dann können wir folgende Funktion schreiben:

FUN *ask*: *String* $\to$ *State* $\to \alpha \times$ *State*
DEF $ask(req)(s) = write(req)(s)\ \&\ \lambda s' \bullet (read(s'),\ s')$

Die Hilfsfunktion $\_\ \&\ \_$ ist dabei folgendermaßen definiert:[3]

FUN $\_\ \&\ \_\colon\ State \times (State \to \alpha \times State) \to (\alpha \times State)$
DEF $(s\ \&\ f) = f(s)$

Damit hat man – oberflächlich betrachtet – eine relativ einfache und leicht verständliche Form gefunden, um aus dem funktionalen Programm heraus den „Weltzustand“ abfragen und sogar ändern zu können: Man macht ihn zum expliziten Parameter.

Aber das bringt auch gravierende Probleme mit sich: Man kann ganz leicht die (physikalisch notwendige) Single-Threadedness verletzen:

FUN *foo*: *State* $\to$ *State*
FUN *bar*: *State* $\times$ *State* $\to$ *State*
DEF *foo* s = *bar*(*write* `"x"` s, *write* `"y"` s) $\quad --$ *Paradoxie!*
DEF $bar(s_1, s_2) = \ldots$

Im Rumpf von *foo* arbeitet die Funktion *bar* mit zwei unterschiedlichen Kopien des Weltzustands; bei der einen steht auf dem Terminal ein `"x"`, bei der

[3] Diese Funktion ist übrigens nichts anderes als die „invertierte Funktionsapplikation“, die wir schon in Kapitel 1 in der Notation $x.f$ benutzt haben.

anderen steht auf dem Terminal ein "y". Das ist ein Widerspruch zu den Gegebenheiten der realen Welt.

Daraus folgt, dass diese Technik nur mit größter Sorgfalt eingesetzt werden darf, wenn das Programm nicht unsinnig sein soll. Und natürlich entspricht es gutem Software-Engineering und damit auch gutem Sprachdesign, die pathologischen Fehlersituationen bereits im Compiler abzufangen.

Die Konsequenzen hatten wir schon in Kapitel 13 diskutiert. Man muss Single-Threadedness garantieren, sei es durch Verwendung Linearer Logik, sei es durch Monaden. In der Praxis durchgesetzt hat sich die Lösung mittels Monaden.

17.2.2 *Continuations + Hiding* = Monaden

Die Continuation-Technik des vorigen Abschnitts löst das Ein-/Ausgabe-Problem ja eigentlich schon recht gut. Es fehlt nur noch die *garantierte* Single-Threadedness, also eine Single-Threadeness, die nicht von der Selbstdisziplin des Programmierers abhängt. Die Lösung ist im Software-Engineering unter dem Stichwort *Hiding* bekannt: Was der Programmierer nicht unkontrolliert manipulieren darf, muss man vor ihm verstecken. Genau das leisten die folgenden Typen und Operationen: Sie verstecken den Zustand.

$$\text{TYPE } M\ \alpha = (State \to \alpha \times State)$$
$$\text{FUN } yield\colon\ \alpha \to M\ \alpha$$
$$\text{FUN } \_\ \&\ \_\colon\ M\ \alpha \times (\alpha \to M\ \beta) \to M\ \beta$$
$$\text{DEF } yield\ a = \lambda s \bullet (a, s)$$
$$\text{DEF } m\ \&\ f = \lambda s \bullet \text{LET } (a,\ s') = m\ s \text{ IN } f\ a\ s'$$

Wenn man dies mit der Funktion _ & _ aus dem vorigen Abschnitt vergleicht, dann ist das Ganze etwas komplexer geworden; aber dafür hat man den *State*-Parameter versteckt. Außerdem wird das gesamte Konzept etwas uniformer und damit das Schreiben größerer Ein-/Ausgabe-Programme ein bisschen einfacher.

Trotzdem bleibt klar erkennbar, dass der Kern dieser Vorgehensweise die Continuation-Technik ist. Mit anderen Worten: Man hat ein Konzept aus der Theorie der Semantik-Definition erfolgreich in die praktische Programmierung übertragen. Dass diese Typen und Funktionen dann auch noch die Monaden-Axiome erfüllen, ist ein erfreuliches, aber eher zufälliges Geschenk.

Somit hat sich die Zustands-Monade aus Abschnitt 11.2.3 im Laufe der letzten Jahre als geeignetes Instrument herauskristallisiert, um Ein-/Ausgabe in funktionale Programme zu integrieren. Ein kleines Detail ist der Unterschied, dass in der Praxis aus Gründen der Effizienz eine einzige Funktion ($State \to \alpha \times State$) benutzt wird, wo wir aus Gründen der Eleganz zwei Funktionen ($State \to \alpha$) und ($State \to State$) verwenden.

17.2.3 Die Ein-/Ausgabe ist eine Compiler-interne Monade

In einer Hinsicht ist die Ein-/Ausgabe-Monade allerdings doch etwas Besonderes: Ihre Operationen lassen sich *nicht* in rein funktionale Ausdrücke einbetten. In Abschnitt 11.2.4 hatten wir uns mit der Einbettung monadischer Ausdrücke in umgebende funktionale Ausdrücke beschäftigt, was folgendermaßen geschrieben werden konnte:

$$\ldots f(\ exec(new(initialState)\ \&\ \ldots\ \&\ \ yield(a))\ )\ \ldots$$

Zur Erinnerung: Der Wert *initialState* liefert den Anfangszustand, mit dem die monadische Berechnung startet. Der letzte monadische Wert – hier a – ist der Wert des gesamten Ausdrucks $exec(\ldots)$. *Aber bei der Ein-/Ausgabe-Monade funktioniert das nicht!* Denn *initialState* müsste hier den initialen „Weltzustand“ liefern und damit wäre die Single-Threadedness schon wieder gefährdet.

Das ist der Hauptgrund, weshalb die Ein-/Ausgabe-Monade nicht vom Benutzer eingeführt wird, sondern vom Compiler vorgegeben ist. (In OPAL ist das z. B. die Monade $Com\ \alpha$, in HASKELL die Monade $IO\ \alpha$.) Konsequenterweise können Ausdrücke vom Typ $Com\ \alpha$ bzw. $IO\ \alpha$ nicht in umfassende Ausdrücke eingebettet werden. Mit anderen Worten: Die Einbettung mittels $exec(new(initialState)\ \&\ \ldots)$ erfolgt hier genau einmal, und zwar implizit durch den Compiler im Augenblick des Programmstarts.

Wir werden auf Fragen der funktionalen Ein-/Ausgabe in Abschnitt 18.2 noch genauer eingehen.

17.3 Zeit: Die elementarste aller Zustands-Monaden

Jetzt wollen wir uns der fundamentalen Frage nach dem Zusammenspiel zwischen der Zeit-losen Welt der funktionalen Programme und der Zeit-basierten Welt der Nutzer dieser Programme zuwenden.

Einen ersten Schritt dazu liefert die Beobachtung, dass „Zeit“ die elementarste aller Zustands-Monaden liefert.[4] Wir gehen von einem vorgegebenen Typ TIME aus, der das intuitive Konzept der physikalischen Zeit reflektiert.

Definition (TIME)

Der vordefinierte Typ TIME ist ein gedankliches Modell der physikalischen Zeit. Auf diesem Typ gibt es nur eine Funktion

```
FUN Δ: TIME → TIME        -- "Fortschreiten" der Zeit
```

Diese Funktion modelliert das Fortschreiten der Zeit, also den Zeitverbrauch der jeweiligen Operation.

Natürlich ist dies nur ein konzeptuelles, gedankliches Modell. Der Typ TIME existiert nirgends real, auch nicht im Compiler. Werte vom Typ TIME

[4] Diese Sichtweise wurde uns von Dusko Pavlovic vorgeschlagen.

kann man nicht anschauen und schon gar nicht manipulieren. Um es nochmals zu betonen: TIME hat nichts mit Uhren zu tun. Letztere existieren in Computern und liefern bei Anfrage Zahlen, von denen man annimmt, dass sie – im Rahmen einer gewissen Genauigkeit – etwas mit der physikalischen Zeit TIME zu tun haben.

Der Spezialtyp TIME ist compilerintern vorgegeben (weshalb wir ihn auch als Schlüsselwort groß schreiben). Es gibt keine Funktion, die ein Ergebnis vom Typ TIME liefert. Somit ist es für den Programmierer technisch unmöglich, Werte vom Typ TIME zu erhalten oder zu manipulieren. Es gibt nur eine einzige – ebenfalls vorgegebene – Funktion auf dem Typ TIME, nämlich die Funktion Δ, mit der das Fortschreiten der Zeit modelliert wird. (Die Eigenschaften von Δ werden in Abschnitt 17.3.1 gleich noch genauer diskutiert.)

Auf dieser Basis können wir eine „Zeit-Monade“ definieren, die eine Instanz *Machine*(TIME) der Zustands-Monade aus Programm 11.6 von Abschnitt 11.2.3 ist. Wegen ihrer Bedeutung geben wir dieser Monade aber eine eigenständige Definition, die im Wesentlichen durch Instanziierung von *Machine*(TIME) entsteht. Das ist in Programm 17.2 angegeben (wobei wir die hier überflüssige Operation *flatten* weglassen).

Programm 17.2 Die Zeit-Monade (1. Versuch)

```
STRUCTURE TimeMonad = {
  PRIVATE TYPE Obs α = (TIME → α)                    -- Hilfstyp
  PRIVATE TYPE Progress = (TIME → TIME)              -- Hilfstyp
  TYPE Beh α = (Obs α × Progress)                    -- "Behaviour"
  FUN _ *: (α → β) → (Beh α → Beh β)                 -- Map
  DEF f * (obs, Δ) = (f ∘ obs, Δ)
  FUN yield: α → Beh α                               -- lift
  DEF yield a = (K a, id)
  FUN _ &: Beh α → (α → Beh β) → Beh β               -- sequ. Komposition
  DEF (obs, Δ) & f = (f ∘ obs) S Δ
                     WHERE
                     (f S g)x = (f x)(g x)           -- S-Kombinator
  FUN _ &: Beh α → Beh β → Beh β                     -- Variante (Kurzform)
  DEF m1 & m2 = m1 & (K m2)
}
```

Wir benutzen zur Schreibabkürzung zwei Hilfstypen: *Obs* α steht für zeitabhängige Werte („*Beobachtungen*“); *Progress* steht für den – nicht näher spezifizierten – Zeitverbrauch einer Operation. Er ist grundsätzlich durch die vorgegebene Funktion Δ spezifiziert.

Der monadische Typ *Beh* α liefert – intuitiv gesprochen – einen zeitabhängigen Wert und den Zeitpunkt, zu dem er vorliegt. Er entspricht dem, was z. B. in OPAL *Com* α heißt und in HASKELL *IO* α.

Die Operation $f * op$ wendet f auf den beobachteten Wert an, verbraucht aber (in unserem konzeptuellen Modell) keine weitere Zeit.

Die Operation *yield a* macht den Wert a zwar zeitabhängig, aber zu jedem Zeitpunkt gleich. Zeit verbraucht dieses formale Lifting nicht. (Wir verwenden zur Definition den K-Kombinator aus Programm 1.1 in Abschnitt 1.2.1.)

Die sequenzielle Komposition *op* & *cont* wendet zuerst die Operation *op* an, wobei ein zeitabhängiger Wert a' beobachtet wird und auch eine gewisse Zeit verstreicht, so dass ein neuer Zeitpunkt $t' = \Delta(t)$ erreicht wird. Die Anwendung $cont(a')$ der Fortsetzungsfunktion liefert eine neue Operation op', die im Zeitpunkt t' ausgeführt wird. Das Ergebnis ist ein Wert a'' und ein neuer Zeitpunkt t''.

Auch hier gibt es die Variante, bei der die zweite Operation direkt gegeben ist und nicht erst aus dem Ergebnis der ersten berechnet werden muss (was wir wieder gut mit dem K-Kombinator aus Programm 1.1 ausdrücken können).

Anmerkung 1: Eigentlich müssten wir die Funktion Δ in der Definition des Operators _ & etwas filigraner fassen. Betrachten wir z. B. ein Programmfragment der Art ... write(file)(text) & read(file) & .... Dann dauert das Schreiben bis zu einem gewissen Zeitpunkt $t' = \Delta(t)$. Die folgende Leseoperation startet aber im Allgemeinen etwas später als t' (z. B. weil das Betriebssystem den Prozess unterbrochen hat), also zu einem Zeitpunkt $t'' = \Delta'(t) > t' = \Delta(t)$. Dieser „Schlupf" müsste eigentlich in der Definition berücksichtigt werden; aber aus Gründen der einfacheren Darstellung verzichten wir auf diese Art von Purismus.

Anmerkung 2: Dieser soeben erwähnte „Schlupf" hat in der realen Welt eine gravierende Auswirkung. Denn es kann passieren, dass parallel laufende Prozesse weitere Änderungen bewirken, so dass (im obigen Beispiel) zum Zeitpunkt des Lesens bereits ein anderer Text in der Datei steht, als der gerade von write geschriebene. (Dieses Problem existiert übrigens auch bei der traditionellen Ein-/Ausgabe-Modellierung mittels der Zustands-Monade; dort ist es aber noch etwas schwieriger zu behandeln als bei unserer Zeit-Monade. Denn die Zeit selbst bleibt von parallelen Prozessen unberührt; wir können allerdings nur noch schwächere Annahmen für die Werte obs(t) machen.)

Anmerkung 3: Paul Hudak benutzt in [81] ebenfalls die Idee von „Zeit", um interaktive Animationen zu programmieren. Dabei wird Time allerdings als Synonym für Float benutzt und in einen Typ der Art TYPE *Animation* $\alpha = (Time \to \alpha)$ *eingebaut. Über diesem Typ werden dann weitere monadische Typen wie Behavior* α *konstruiert, die als Basis für die Animationen dienen. Die Grundlage dafür ist ein Konzept, das andernorts auch unter dem Namen* Time-tagged event streams *firmiert.*

17.3.1 Zeitabhängige Operationen und Evolution

Wie wir schon in Kapitel 11 gesehen haben, stellt die Definition der Basisoperationen das größte Problem bei den Zustands-Monaden dar. Deshalb müssen wir uns mit diesem Problem auch bei der Zeit-Monade beschäftigen. Wir

illustrieren dies anhand von einfachen Ein-/Ausgabe-Operationen. (Die tatsächliche Diskussion der Ein-/Ausgabe-Operationen wird allerdings erst in Abschnitt 18.2 erfolgen.)

Was geschieht, wenn wir die Ein-/Ausgabe, also die *IO*-Monade, nicht als allgemeine Zustands-Monade, sondern über die Zeit-Monade modellieren?

```
TYPE IO α = Beh α   -- Ein-/Ausgabe als Zeit-Monade
```

Der Einfachheit halber betrachten wir vorläufig nur Textdateien und für diese auch nur die beiden Operationen *read* und *write*, die den ganzen Inhalt als *String* lesen bzw. schreiben. Dies reicht aus, um die wesentlichen Aspekte zu diskutieren. *Was ist dann eine Datei?* Antwort: Etwas, was einen String als Inhalt hat, allerdings zu verschiedenen Zeitpunkten verschiedene Strings. Deshalb ist eine Datei eine Abbildung von Zeitpunkten auf Strings.

```
TYPE File = (TIME → String)
```

Die monadische Operation *read*(*file*) beschafft zu einem gegebenen Zeitpunkt t den aktuellen Inhalt der Datei. Der Beobachtungsteil ist also nichts anderes als der aktuelle Wert der Funktion *file* selbst; dazu kommt ein Zeitverbrauch Δ.

```
FUN read: File → IO String
DEF read(file)(t) = (file(t), Δ(t))
```

Die Aktion *write*(*file*)(*text*) hat keinen Wert, aber einen Effekt: Sie „ändert“ den zeitlichen Wert von *file*. Nehmen wir an, *write* wird zu einem Zeitpunkt t ausgeführt. Dann hat ab einem gewissen Zeitpunkt $t' = \Delta(t)$ die Funktion *file*(t') den neuen Wert *text*, den sie bis zu einem Zeitpunkt t'' behält, an dem das nächste *write* einen neuen Inhalt festlegt. Diesen Aspekt drücken wir durch das Schlüsselwort EVOLVE aus (von dem wir in Kapitel 18 ausgiebig Gebrauch machen werden).

```
FUN write: File → String → IO Void
DEF write(file)(text)(t) = (void, Δ(t))
                     EVOLVE file(Δ(t)) = text
```

Den Zusammenhang zwischen dem Zeitverlauf Δ, den zeitabhängigen Funktionen wie *file* und dem EVOLVE-Konstrukt müssen wir noch genauer diskutieren. Vielleicht hilft es dabei der Intuition, wenn wir Douglas Adams ins *Milliways* begleiten. Dort, im *Restaurant am Ende des Universums* [12], sind wir am Ende der Zeit und blicken zurück auf den gesamten Lauf der Geschichte. Und dann würde sich uns ein Bild wie in Abbildung 17.4 bieten.

Die horizontale Achse repräsentiert den Lauf der Zeit und die vertikale Achse die (ungeordnete) Menge aller möglichen Dateiinhalte, in unserem Beispiel also alle möglichen Texte. Die Zeit zerfällt – bezogen auf die Datei *file* – in Intervalle, in denen die Datei jeweils einen gewissen Text als Inhalt hat.

Wie diese Intervalle liegen, hängt von den Zeitpunkten ab, an denen die Operation *write* jeweils ausgeführt wird. Weil wir auf diese Zeitpunkte keinen

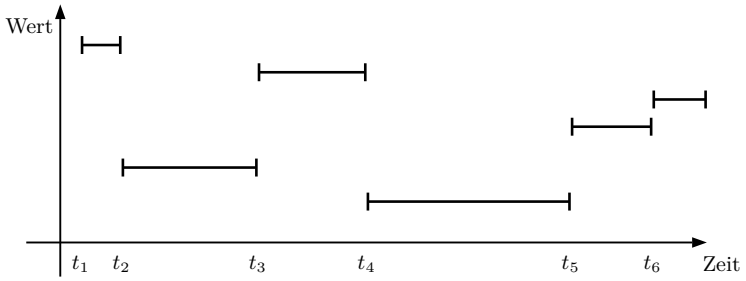


Abb. 17.4: Die „Funktion“ *file*

direkten Einfluss haben, können wir den Zeitverbrauch in der Zeit-Monade nur mittels einer impliziten Funktion Δ modellieren, über die wir nichts weiter wissen, als dass sie monoton wächst. Das heißt, Δ ist eine feste, vorgegebene Funktion. Den tatsächlichen Wertverlauf dieser Funktion kennt man nur a posteriori, also nach Beendigung des jeweiligen Programmlaufs. (Man kann Δ auch als eine Art von globalem „Orakel“ auffassen, das die jeweils nächsten Zeitpunkte voraussagt.)

Damit bleibt die Rolle von EVOLVE zu klären.

Definition (EVOLVE)

Für monadische Operationen, die auf zeitabhängigen Funktionen der Art

FUN f: TIME $\to \alpha$

arbeiten, kann bei der Definition eine EVOLVE-Klausel angegeben werden:

FUN op: (TIME $\to \alpha$) $\to$ *Beh* β
DEF $op(f)(t) = \ldots$ EVOLVE $f(\Delta(t)) = \langle\!\langle\, Wert \,\rangle\!\rangle$

Mit dieser Klausel werden Constraints für den Wertverlauf der Operation f über die Zeit hinweg festgelegt. Es gibt auch eine schönere syntaktische Variante, bei der kein explizites Zeitargument gebraucht wird:

DEF $op(f) = \ldots$ EVOLVE $f \rightsquigarrow \langle\!\langle\, Wert \,\rangle\!\rangle$

Was bedeutet dies konzeptuell? Eine zeitabhängige Funktion wie das obige Beispiel *file* wird im Laufe der Zeit (modelliert durch die globale Funktion Δ) immer wieder in Operationen wie *write* benutzt. Damit liefern die zugehörigen EVOLVE-Klauseln eine *induktive Definition* des Wertverlaufs von *file*. Es ist die Aufgabe des Compilers sicherzustellen, dass die interne Implementierung dieser induktiven Definition genügt.

17.3.2 Zeit-Monade oder Zustands-Monade?

Auf den ersten Blick sieht die Zeit-Monade nicht viel anders aus als die übliche Zustands-Monade. Aber bei genauerem Hinsehen zeigen sich eine Reihe konzeptueller Vorteile.

- Die „Zeit“ ist ein klares, einfaches und universelles Konzept. Es gibt keine Notwendigkeit für einen „Mega-State“, der ambivalent und nebulös bleiben muss, weil man bei Bedarf noch alles hineinpacken muss, was im Umfeld eines Programms auftauchen kann.
- Die Alternative zu diesem allumfassenden Mega-State besteht darin, Monaden mit unterschiedlichen Instanzen von *State* zu kombinieren. Aber die Kompositionalität von Monaden mit unterschiedlichem *State* ist nicht problemlos. (Sie werden in der Literatur unter dem Stichwort *Monaden-Transformer* diskutiert.) In Abschnitt 11.2.4 hatten wir sie ziemlich ad hoc mit einer sehr unbefriedigenden automatischen Produktbildung behandelt. Dieses Problem löst sich jetzt von selbst: Es gibt nur noch TIME als internen Zustand.
- Die Single-Threadness kann schon alleine deshalb nicht mehr verletzt werden, weil wir mangels einer Operation *initialTime* Zeit-monadische Ausdrücke nie in normale funktionale Ausdrücke einbetten können.
- Das Konzept des automatischen Monaden-Liftings, das bereits in Abschnitt 11.2.4 angesprochen und im *Eval*-Beispiel in Abschnitt 17.1.1 nochmals stark motiviert wurde, ist bei einer einzigen Art von Monade (nämlich der im Compiler vordefinierten Zeit-Monade) viel einfacher zu realisieren, als bei beliebigen benutzerdefinierten Monaden.
- Wir können das Software-Engineering-Prinzip der Modularisierung beibehalten, weil wir alle benötigten Strukturen und Objekte explizit im Programm auflisten und in entsprechende Pakete und Bibliotheken einordnen können.
- Die Verwendung der Zeit zumindest als konzeptuelles Modell erlaubt es, Basisoperationen klarer zu spezifizieren (auch wenn ihre Implementierung nach wie vor im Compiler häufig durch direkten Zugriff auf Betriebssystem-Dienste realisiert werden muss). Mit dem Konzept der EVOLVE-Klauseln kann man präzise festlegen, welche Zustandsänderungen zu erfolgen haben.
- Begriffe wie *File* sind als Abbildungen von Zeiten auf Inhalte wesentlich abstrakter als ominöse Referenzen auf „Handles“ in Betriebssystemen. Dieses Konzept ist daher wesentlich adäquater für die Funktionale Programmierung. (Wir werden gleich noch sehen, dass sich damit das ganze Paradigma der objektorientierten Programmierung in unsere funktionale Welt integrieren lässt.)
- Nicht zuletzt eröffnet ein solcher zeitbasierter Ansatz die Chance, auch parallele Prozesse sauber in die Funktionale Programmierung zu integrieren. (Darauf kommen wir – zumindest skizzenhaft – später noch einmal zurück.)

17.4 Die erweiterte Zeit-Monade

Die Struktur *TimeMonad* in Programm 17.2 ist nur eine Instanz der Zustands-Monade, die die elementaren Monaden-Operationen enthält. Für das praktische Arbeiten sind aber noch weitere Konzepte und Operationen notwendig, die wir im Folgenden vorstellen wollen. (Eine zusammenfassende Präsentation der kompletten Struktur findet sich in Programm 17.3 am Ende dieses Kapitels.)

17.4.1 Exceptions

Es liegt in der Natur der Sache, dass bei der Interaktion mit der Umgebung *Ausnahmesituationen* (engl.: *Exceptions*) entstehen können, die sich nicht durch vorherige IF-Abfragen abfangen lassen. Ein typisches Beispiel ist das Lesen von einer Datei. Theoretisch könnte man das Lesen in eine Abfrage der folgenden Bauart einbetten (unter der Annahme, dass ein entsprechendes monadisches IF existiert):

```
... IF exists(file) THEN ... read(file) ...   -- klappt nicht!
```

Das Problem ist hier, dass zwischen dem Test *exists*(*file*) und dem Lesen *read*(*file*) ein anderer Prozess die Datei gelöscht haben könnte. Deshalb ist das potenzielle Scheitern bei Operationen wie *read* ein unvermeidbares, inhärentes Phänomen; denn diese Operationen interagieren mit der Umgebung, die sich unserer Kontrolle vollständig entzieht.

Um mit diesem Problem umzugehen, sind die Ein-/Ausgabe-Monaden in Bibliotheken wie der BIBLIOTHECA OPALICA etwas aufwendiger implementiert, als wir das bisher skizziert haben. Diese Erweiterungen müssen wir auch bei uns vornehmen, wenn wir ein praktikables Konzept entwickeln wollen. Vor allem müssen wir bei den beobachteten Werten die Möglichkeit des Scheiterns vorsehen. Dazu gibt es den Typ *Maybe* (vgl. Abschnitt 8.1).

```
STRUCTURE TimeMonad = {
  PRIVATE TYPE Obs α = (TIME → α)                 -- Hilfstyp
  PRIVATE TYPE Progress = (TIME → TIME)           -- Hilfstyp
  TYPE Beh α = (Obs(Maybe α) × Progress)          -- "Behaviour"
  ...
  DEF (op & f)(t) = LET (a, t') = op(t)
                    IN
                    IF a: Fail THEN (a, t')       -- Fehler durchreichen
                               ELSE f(a)(t') FI
}
```

Beobachtungen liefern jetzt entweder einen gültigen Wert oder *fail*. Deshalb müssen alle monadischen Operationen eine entsprechende Fallunterscheidung vorsehen (s. Programm 17.3 auf Seite 379). Die Operatoren $*$ und *yield*

sind trivial erweiterbar. Interessanter ist der Operator $(op \,\&\, f)$. Er wird so adaptiert, dass er Fehler automatisch propagiert: Wenn die Ausführung des ersten Kommandos *op* auf einen Fehler führt, also den Wert *fail* liefert, dann wird die folgende Operation f gar nicht mehr ausgeführt, sondern sofort das *fail* durchgereicht; bei wohl definierten Werten wird f dagegen wie üblich ausgeführt.

Mit dem so adaptierten &-Operator hat ein Programm jetzt folgendes Verhalten: Sobald irgendwo im Laufe der Ausführung ein Fehler auftritt, wird er bis zum Programmende durchgereicht. Mit anderen Worten, *ein Fehler führt zum sofortigen Programmabbruch.*

Eine solche Brute-force-Methode ist aber nicht generell akzeptabel. Also muss man eine Möglichkeit schaffen, das vollständige Durchreichen der Fehler zu unterbrechen. (In Sprachen wie JAVA lässt sich dies durch die `try-catch`-Konstrukte bewerkstelligen.) Dazu führen wir einen speziellen Operator „ // “ ein:

```
FUN _ // _: Beh α × (Fail → Beh α) → Beh α
DEF (try // catch)(t) =
                    LET (a, t') = try(t)
                    IN
                    IF a: Fail THEN catch(a)(t')   -- Misserfolg retten
                              ELSE (a, t')      FI -- Erfolg durchreichen
```

Der *Catch*-Operator $(try \;//\; (\lambda x \bullet catch(x)))$ führt zuerst die Operation *try* aus. Falls dies erfolgreich ist, wird *catch* ignoriert und das Ergebnis von *try* als Ergebnis des ganzen Ausdrucks durchgereicht. Falls *try* jedoch auf einen Fehler läuft, wird dieser an die Operation *catch* übergeben, die dann gegebenenfalls Reparaturversuche unternehmen kann.

In der Praxis muss man die Fehlersituationen allerdings filigraner erfassen, als dies mit unserem einfachen Typ *Maybe* α möglich ist. Dazu genügt es, die Variante *Fail* dieses Typs entsprechend zu erweitern:

```
TYPE Fail  = { fail } | Exception
TYPE Exception = String | ...
```

Der Typ *Exception* beschreibt die Menge aller möglichen Exceptions. Er ist rein technischer Natur und daher für uns hier nicht weiter interessant. (Wer wissen möchte, wie so ein Typ in der Praxis aussieht, kann z. B. in der Bibliothek von JAVA nachsehen.) Um einfache Fälle bequem handhaben zu können, nehmen wir allerdings schlichte Strings („Fehlermeldungen“) als eine Möglichkeit der Exceptiondarstellung mit auf.

Im Interesse einer konzisen Notation führen wir auch noch eine monadische Operation ein, mit der Exceptions ausgelöst werden können (also das Gegenstück von `throw «exception»` in JAVA). Wir überlagern dazu den Namen *fail*, der im Typ *Maybe* α verwendet wird.

```
FUN fail[α]: Exception → Beh α
DEF fail[α](e)(t) = (e, t)              -- "yield" Exception e
```

Man beachte die etwas trickreiche Typisierung. Wenn der Aufruf $fail(e)$ in einem Kontext steht, in dem z. B. eine Operation der Art $Beh\ Int$ erwartet wird, dann wird α durch diesen Kontext zu Int instanziiert. Da der Beobachtungsteil von $fail(e)$ den Typ $Maybe\ Int$ hat, ist die Exception e hier typkorrekt.

17.4.2 Choice

Wir wollen hier auch noch einen weiteren Operator vorstellen, der seine große Nützlichkeit erst im Zusammenhang mit den parallelen Agenten im Kapitel 19 entfalten wird. Aber es gibt auch ohne Parallelität einige Situationen, in denen dieser Operator gut brauchbar ist.

Ein typisches Beispiel findet sich im Bereich von Kontrollsoftware. Hier hat man oft ein zyklisches Abfragen von Sensoren, wobei nur bei Vorhandensein eines Signals entsprechende Aktionen auszulösen sind. Das sieht dann typischerwiese so aus wie in dem folgenden Programmfragment:

$$
\begin{array}{l}
\ldots \\
(\quad (read\ sensor_1\ \&\ action_1) \\
+\ (read\ sensor_2\ \&\ action_2) \\
+\ \ldots \\
+\ (read\ sensor_n\ \&\ action_n)\ ) \\
\ldots
\end{array}
$$

Dazu brauchen wir einen entsprechenden Auswahloperator „ + “:

$$
\begin{array}{l}
\text{FUN}\ \_ + \_ \colon\ Beh\ \alpha \times Beh\ \alpha \to Beh\ \alpha \\
\text{DEF}\ (op_1 + op_2)(t) = \text{«}wähle\ erstes\ Kommando,\ das\ \texttt{"}bereit\texttt{"}\ ist\text{»}
\end{array}
$$

Dabei gibt es allerdings ein subtiles Problem mit dem Begriff „bereit“.[5] Auf der Implementierungsebene haben die atomaren Basisoperationen der Zeit-Monade – also z. B. *read*, *write*, *open* etc. – zusätzlich noch die Eigenschaft „bereit“ zu sein (was wir auf unserer konzeptuellen Ebene bisher nicht mitmodelliert haben). In einer Auswahl wird dasjenige Kommando genommen, dessen erstes Basiskommando „zuerst bereit“ ist.

Auf der Basis der Zeit-Monade ist das ein relativ naheliegendes und auch einfach beschreibbares Konzept. Aber es gibt einen kleinen Fallstrick. Die Kommandos einer Auswahl müssen in irgendeiner Reihenfolge auf „Bereitschaft“ geprüft werden – und dieser Vorgang kostet selbst Zeit. Wenn dann zwei Kommandos „fast gleichzeitig“ bereit werden, dann kann es passieren, dass das spätere vor dem früheren entdeckt wird. (Man spricht dann auch von *Race condition.*) Um hier unnötige und ineffiziente Zusatztests zu vermeiden, muss man den Begriff „gleichzeitig“ so fassen, dass er Raum für diese Unschärfe lässt.

[5] Diese Frage hat auch lange die Semantikdefinition von ADA belastet.

17.4.3 Die Systemuhr und Timeouts

Im Zusammenhang mit dem Auswahloperator, spätestens aber mit der Einführung von parallelen Prozessen (s. Kapitel 19), entsteht manchmal das Bedürfnis, eine Rechnung für eine gewisse Zeit ruhen zu lassen. Dazu führen wir eine entsprechende Operation *timeout* ein.

```
FUN timeout: Nat → Beh Void
DEF timeout(n)(t) = (void, Δ(t))   -- Prozess "schläft" n Millisekunden
```

Diese Operation liefert kein Ergebnis und hat auch keinen sichtbaren Effekt. Sie verbraucht nur Zeit.

Das sieht zwar sehr einfach und naheliegend aus, hat aber einen interessanten philosophischen Aspekt. Hier taucht zum ersten (und einzigen) Mal eine Beziehung zwischen unserer konzeptuellen Zeit TIME und der Zeit *Time* der Systemuhren auf. Letztere ist ein beobachtbarer Wert, den wir der Einfachheit halber als natürliche Zahl auffassen, die als Dimension Millisekunden repräsentiert. Mit der Operation *timeout* ist dann die Hoffnung verbunden, dass die reale Zeitdifferenz $\Delta(t) - t$ ungefähr n Millisekunden entspricht. Inwieweit diese Hoffnung berechtigt ist, hängt von der Implementierung des Compilers und der Hardware der zugrunde liegenden Maschine ab.[6]

17.4.4 Zusammenfassung: Die Zeit-Monade

Wegen der fundamentalen Rolle der Zeit-Monade lohnt es sich, ihre Definition noch einmal vollständig zusammenzufassen. Das ist in Programm 17.3 gezeigt. Gegenüber der einfachen Variante von Programm 17.2, die im Wesentlichen nur die klassischen Monaden-Operationen enthält, umfasst diese Version von *TimeMonad* noch eine Reihe von zusätzlichen Operationen, die in der Praxis nützlich sind.

- Der wichtigste Aspekt der Erweiterung ist die Umstellung der beobachteten Werte auf den Typ *Maybe* α. Damit lassen sich Ausnahmesituationen systematisch erfassen.
- Die Operation *forever* führt eine gegebene Operation immer wieder aus. Dies ist bei „normalen“ monadischen Operationen nicht sehr wichtig, wird sich aber im Zusammenhang mit parallelen Agenten in Kapitel 19 als durchaus nützlich erweisen.
- Die Operation *done* ist an den Stellen hilfreich, an denen man aus formalen Gründen noch eine monadische Operation braucht, obwohl nichts mehr zu tun ist.
- Der Auswahloperator _ + _ wird seine Nützlichkeit auch erst im Zusammenhang mit parallelen Agenten entfalten.

[6] Aus dem Gebiet der Realzeitprogramme und der Betriebssysteme ist wohl bekannt, dass diese Hoffnung nicht immer berechtigt ist.

Programm 17.3 Die vollständige Zeit-Monade

```
STRUCTURE TimeMonad = {
  PRIVATE TYPE Obs α = (TIME → α)                          -- Hilfstyp
  PRIVATE TYPE Progress = (TIME → TIME)                    -- Hilfstyp
  TYPE Beh α = (Obs(Maybe α) × Progress )                  -- "Behaviour"
  FUN _ *: (α → β) → (Beh α → Beh β)                       -- Map
  DEF f * (obs, Δ)(t) = LET a = obs(t) IN
                        IF a: Fail THEN (a, Δ(t))          -- Fehler durchreichen
                                   ELSE (f(a), Δ(t)) FI
  FUN yield: α → Beh α                                     -- lift
  DEF (yield a)(t) = (a, t)
  FUN _ &: Beh α → (α → Beh β) → Beh β                     -- sequ. Komposition
  DEF (op & f)(t) = LET (a, t') = op(t) IN
                    IF a: Fail THEN (a, t')                -- Fehler durchreichen
                               ELSE f(a)(t') FI
  FUN _ &: Beh α → Beh β → Beh β                           -- Variante (Kurzform)
  DEF m1 & m2 = m1 & (K m2)
  FUN forever: Beh α → Beh α
  DEF forever(op) = op & forever(op)                       -- & ist lazy!
  FUN done: Beh Void
  DEF done = yield void
  FUN _ + _: Beh α × Beh α → Beh α
  DEF (op1 + op2)(t) = «wähle erstes Kommando, das "bereit" ist»
  FUN _ // _: Beh α × (Fail → Beh α) → Beh α
  DEF (try // catch)(t) = LET (a, t') = try(t) IN
                          IF a: Fail THEN catch(a)(t')     -- Misserfolg retten
                                     ELSE (a, t') FI       -- Erfolg durchreichen
  FUN fail[α]: Exception → Beh α
  DEF fail[α](e)(t) = (e, t)                               -- "yield" Exception e
  FUN timeout: Nat → Beh Void
  DEF timeout(n)(t) = (void, Δ(t))                         -- "schläft" n Millisekunden
}
```

- Das Abfangen von Exceptions gehört zum wesentlichen Repertoire jedes Programms, das mit der Umgebung interagiert. Dabei ist es manchmal angenehm, auf kompakte Weise selbst Exceptions auslösen zu können; dies ermöglicht die Operation *fail*.
- In einigen Situationen muss man die Möglichkeit haben, einen Prozess für eine gewisse Zeit zu verzögern; das leistet die Operation *timeout*.

18

Objekte und Ein-/Ausgabe

Die „Tücke“ des Objekts ist ein dummer Anthropomorphismus.
Wittgenstein

Wir haben bereits den Erfolg des *objektorientierten Paradigmas* angesprochen. Dieses Paradigma wird gemeinhin mit zwei Eigenschaften identifiziert: *Objektkonzept* und *Vererbung*. Letzteres haben wir in den Kapiteln 5 und 7 schon detailliert im Zusamenhang mit Gruppenmorphismen und Subtypen diskutiert. Mit anderen Worten: Vererbung ist bereits ein Bestandteil unserer funktionalen Welt. Damit bleibt das Objektkonzept.

18.1 Objekte als zeitabhängige Werte

Um die Diskussion etwas griffiger zu machen, betrachten wir ein kleines Beispiel in einer traditionellen objektorientierten Sprache. Die Idee eines Punktes im $\mathbb{R}^2$ lässt sich z. B. in JAVA (im Wesentlichen) schreiben wie in Abbildung 18.1.

```
class Point {                                                  // JAVA!
   float x;
   float y;
   Point (float x, float y) { this.x = x; this.y = y; }
   float dist () { return sqrt(x*x + y*y); }
   void shift (float dx, float dy) { x += dx; y += dy; }
}
```

Abb. 18.1: Eine JAVA-Klasse

Ein neuer Punkt wird durch einen Aufruf der Konstruktormethode erzeugt, z. B. in der Form `Point p = new Point(3,4)`. Die anderen Methoden lassen

sich dann über die so genannte Punktnotation aufrufen, also z. B. `p.dist()` oder `p.shift(1,-1)`.

Die Übertragung der typischen Paradigmen der Objektorientierung – insbesondere die Sicht von Objekten als unabhängig arbeitenden, persistenten Entitäten, die mit anderen Objekten interagieren – in unsere funktionale Welt stößt an einigen Stellen auf subtile technische Schwierigkeiten. Wir wollen aber der Versuchung widerstehen, diese Probleme durch die sofortige Einführung von Ad-hoc-Notationen „wegzudefinieren". Stattdessen werden wir die Einbettung zunächst ganz puristisch vornehmen (und die dazu notwendigen Komplikationen in Kauf nehmen). Danach werden wir lesefreundlichere Schreibabkürzungen einführen – die dann aber für klar definierte funktionale Konzepte stehen.

Da im Zentrum der folgenden Diskussionen der Objektbegriff steht, wollen wir ihn als Erstes ganz formal fassen.

Definition (Objekt)

Da *Objekte* ihren Zustand ändern können, sind sie zeitabhängige Werte. Deshalb führen wir einen entsprechenden polymorphen Typ als Schreibabkürzung ein.

TYPE $Obj\ \alpha = (\text{TIME} \to \alpha)$ $\quad --$ *Objekte sind zeitabhängige Werte*

Von den typischen Paradigmen objektorientierter Sprachen wie JAVA lassen sich zwei Aspekte in unserem Kontext trivial behandeln:

- JAVA-Klassen dienen simultan der Modularisierung und der Einführung neuer Typen. Dazu gibt es bei uns die getrennten Konzepte von Strukturen und Typen.

  STRUCTURE *Point* = {
    TYPE *Point* = ...
          ...
  }

  Das Overloading des Namens *Point* sollte dabei weder für den Programmierer noch für den Compiler ein Problem darstellen.
- Die Punktnotation ist ebenfalls kein Problem. Wir haben den entsprechenden Operator „_._" schon in Kapitel 1 kennengelernt. Die Aufrufe der Art $p.dist()$ und $p.shift(dx, dy)$ entsprechen also Aufrufen der Form $dist(p)$ und $shift(p)(dx, dy)$ – was übrigens auch von JAVA-Compilern intern so realisiert wird.

Ein anderer Aspekt stellt uns dagegen vor größere Probleme: Wenn wir Objekte in Sprachen wie JAVA betrachten, dann gehören zu ihnen (zumindest konzeptuell) nicht nur die Attribute, sondern auch die Methoden. Das würde in unserer Welt bedeuten, dass im Beispiel aus Abbildung 18.1 der Typ *Point*

nicht nur die x- und y-Komponente umfasst, sondern die *ganze Struktur*, einschließlich aller Operationen. Das ist bei uns auch prinzipiell möglich, weil wir Strukturen ja als *First-class citizens* auffassen (s. Kapitel 4). Allerdings stoßen wir dabei auf einige technische Schwierigkeiten, die das Programm etwas aufwendig machen.

Im Programm 18.1 ist das Prinzip, dass *Objekte ganze (zeitabhängige) Strukturen* sind, umgesetzt. Die Einzelteile dieses Konzepts spielen auf subtile Weise miteinander zusammen, weshalb wir sie im Folgenden Punkt für Punkt diskutieren. (Außerdem legt die relativ aufwendige – aber immer gleiche – Konstruktion die Einführung entsprechender notationeller Abkürzungen nahe, die wir gleich anschließend in Abschnitt 18.1.1 präsentieren werden.)

Programm 18.1 Punkte als Objekte

```
SIGNATURE PointSig = {                          -- Typ der Objekte
  FUN x: Beh Real
  FUN y: Beh Real
  FUN dist: Beh Real
  FUN shift: Real × Real → Beh Void
}

STRUCTURE Point = {
  TYPE Point = Obj PointSig                     -- gleichwertig zu (TIME → PointSig)
  FUN point: Real × Real → Beh Point            -- Konstruktor
  DEF point(x0, y0)(t0) = (p, Δt0)
      WHERE
      p(t0) = { DEF x = yield x0
                DEF y = yield y0
                DEF dist = yield √(x0² + y0²)
                DEF shift(dx, dy) = yield void
                                    EVOLVE p ⇝ point(x0 + dx, y0 + dy)
              }
  FUN x: Point → Beh Real
  DEF x(p)(t) = (p(t).x)(t)
  FUN y: Point → Beh Real
  DEF y(p)(t) = (p(t).y)(t)
  FUN dist: Point → Beh Real
  DEF dist(p)(t) = (p(t).dist)(t)
  FUN shift: Point → Real × Real → Beh Void
  DEF shift(p)(dx, dy)(t) = (p(t).shift(dx, dy))(t)
}
```

Am auffallendsten ist hier, dass die meisten Operationen zweimal vorkommen, wenn auch in leicht unterschiedlicher Form. Der Grund für dieses

Phänomen wird sich in der folgenden Diskussion zeigen. (Hier liegt aber auch die Hauptmotivation für die spätere Einführung von Spezialnotationen.)

- Um alle Teile des Konzepts sauber ausdrücken zu können, müssen wir die Struktur, für die das Objekt steht, typisieren können. Deshalb brauchen wir eine entsprechende Spezifikation oder zumindest Signatur, die hier mit *PointSig* bezeichnet wird. (Dem entspricht in JAVA am ehesten das Konzept der *Interfaces*.)
- Mit Hilfe dieser Signatur können wir den Typ *Point* für Punkte einführen: Punkte sind Objekte, die zu jedem Zeitpunkt eine Struktur der Art *PointSig* repräsentieren.
- Punkte müssen erzeugt werden; dazu führen wir eine Funktion $point(x_0, y_0)$ ein. (Diese Funktion entspricht den *Konstruktormethoden* von JAVA.)
  Da Punkte zeitabhängige Werte sind, muss ihre Erzeugung durch eine monadische Operation erfolgen. Deshalb hat $point(\ldots)$ den Resultattyp *Beh Point*. Wenn z. B. der monadische Konstruktor $point(3, 4)$ zum Zeitpunkt t_0 ausgeführt wird, ist das Resultat p ein Objekt, also eine Funktion der Art (TIME $\to$ *PointSig*). Diese Funktion $p = point(3, 4)$ wird durch zwei Eigenschaften definiert:
  1. Die erste Eigenschaft legt den Funktionswert zum Anfangszeitpunkt t_0 fest. Zu diesem Zeitpunkt ist ihr Wert die folgende Struktur:

$$
\begin{aligned}
p(t_0) = \{\ & \text{DEF } x = \mathit{yield}\ 3 \\
& \text{DEF } y = \mathit{yield}\ 4 \\
& \text{DEF } \mathit{dist} = \mathit{yield}\ \sqrt{3^2 + 4^2} \\
& \text{DEF } \mathit{shift}(dx, dy) = \ \mathit{yield}\ \mathit{void} \\
& \qquad\qquad\qquad\qquad \text{EVOLVE } p \rightsquigarrow \mathit{point}(3 + dx, 4 + dy) \\
& \}
\end{aligned}
$$

  2. Die zweite Eigenschaft beschreibt, wie sich der Funktionswert induktiv zu den weiteren Zeitpunkten ergibt. Dies wird – wie in Abschnitt 17.3.1 beschrieben – durch die EVOLVE-Klauseln festgelegt (von denen es hier nur eine gibt, nämlich bei *shift*). Wenn die Operation *shift* zu einem Zeitpunkt t ausgeführt wird, dann hat p zum Zeitpunkt $t' = \Delta t$ als Wert die mittels $point(\ldots)$ generierte neue Struktur.
- Die Beobachtungsoperationen $x(p)$, $y(p)$ und $dist(p)$ erläutern wir am Beispiel $dist(p)$ für den Punkt $p = point(3, 4)$. Wenn die monadische Operation $dist(p)$ zum Zeitpunkt t ausgeführt wird, müssen wir zuerst den Wert des Objekts p zum Zeitpunkt t bestimmen; dies ist eine Struktur vom Typ *PointSig*. Aus dieser Struktur wählen wir jetzt die Operation *dist* aus. (Man beachte das Overloading des Funktionsnamens!) Diese Operation ist wieder monadisch, nämlich vom Typ *Beh Real*. Also müssen wir sie zum Zeitpunkt t anwenden. Das Ergebnis ist das Paar $(5, t') = (\sqrt{3^2 + 4^2}, \Delta t)$.
- Für die Operation $\mathit{shift}(p)(dx, dy)$ gilt das Analoge. Aber jetzt kommt noch hinzu, dass das Verhalten des Objekts (also der zeitabhängigen Funktion) p weiter festgelegt wird: Zum „nächsten“ Zeitpunkt Δt hat $p(\Delta t)$

als Wert eine neue Struktur. Diese ist das Resultat des Konstruktors $point(p(t).x + dx,\ p(t).y + dy)$.

Zur Illustration dieser Definition betrachten wir ihre Verwendung in einem kleinen Programmfragment.

$$
\begin{array}{llll}
\ldots & \& & point(3,4) & \& \lambda p \bullet \\
 & & p.dist & \& \lambda d_1 \bullet \quad -- \ d_1 = 5.0 \\
 & & p.shift(1,1) & \& \\
 & & p.dist & \& \lambda d_2 \bullet \quad -- \ d_2 = 6.4 \\
 & & \ldots &
\end{array}
$$

Der Name p steht für die schon mehrfach erwähnte zeitabhängige Struktur, die zunächst (zum Zeitpunkt t_1) die Attributwerte 3 und 4 und die entsprechenden Operationen umfasst. Der Aufruf $p.dist$ wertet (zu diesem Zeitpunkt t_1) die Operation *dist* aus dieser anfänglichen Struktur aus; das Ergebnis 5.0 wird an den Namen d_1 gebunden (und die Zeit schreitet zu $t_2 = \Delta t_1$ fort). Die Aktion $p.shift(1,1)$ schreitet zum Zeitpunkt $t_3 = \Delta t_2$ fort und legt implizit fest, dass die Funktion p zu diesem Zeitpunkt eine entsprechend geänderte Struktur repräsentiert. Wenn die Operation $p.dist$ zu diesem Zeitpunkt t_3 aufgerufen wird, ist das Ergebnis jetzt 6.4 (und die Zeit schreitet zu $t_4 = \Delta t_3$ fort). Und so weiter.

Weshalb ist die aufwendige Programmierung im Stil von Programm 18.1 notwendig? Zunächst sind die Operationen der Struktur *Point* in dieser Form nötig. Denn da wir Operationen wie $p.dist$ oder $p.shift(\ldots)$ in Programmen aufrufen, müssen sie den entsprechenden Punkt als explizites Argument haben. Andererseits können wir die Abhängigkeiten aller Operationen von den Attributwerten x und y (die im Konstruktor gesetzt werden) nur ausdrücken, indem wir alle Operationen in einer gemeinsamen, entsprechend parametrisierten Struktur definieren.[1] In diesen Operationen darf dann allerdings der Punkt nicht noch einmal als Argument erscheinen (weil er ja schon die ganze Struktur selbst ist).

Im Übrigen sei darauf hingewiesen, dass unser Ansatz extrem flexibel ist. Wir haben zwar im Beispiel der Operation *shift* nur eine Änderung der Attribute vorgeführt. Aber es wäre genauso gut möglich, dass bei der Evolution eine gänzlich andere Struktur entsteht, bei der die Operationen *dist*, *shift* etc. völlig neue Definitionen erhalten. Die einzige Anforderung ist, dass der Typ *PointSig* erhalten bleibt. Auch hier ist also die Flexibilität objektorientierter Sprachen wie SMALLTALK nachbildbar.

18.1.1 Spezielle Notationen für Objekte und Klassen

Der Ansatz, der im Beispiel von Programm 18.1 vorgeführt wird, ist relativ komplex und verlangt subtil aufeinander abgestimmte Funktionen. Das ist für

[1] Compilertechnisch müssen solche Konstruktionen über so genannte *Closures* realisiert werden; in denen sind die Werte x und y dann implizit versteckt.

die praktische Programmierung ein schwerwiegendes Defizit.[2] Deshalb liegt es nahe, spezielle Notationen einzuführen, mit denen sich die Programmierung kompakter und weniger fehleranfällig gestalten lässt. Wichtig ist dabei aber, dass es sich tatsächlich nur um notationelle Abkürzungen handelt, so dass nicht unter der Hand neuartige, nicht-funktionale Konzepte eingeführt werden.

Programm 18.2 zeigt, wie sich das Beispiel aus Programm 18.1 kompakter schreiben lässt, wenn man geeignete neue Schlüsselwörter einführt.

Programm 18.2 Die Klasse *Point*

```
CLASS Point(x0: Real, y0: Real) = {
  FUN x: Beh Real
  DEF x = yield x0
  FUN y: Beh Real
  DEF y = yield y0
  FUN dist: Beh Real
  DEF dist = yield √(x0² + y0²)
  FUN shift: Real × Real → Beh Void
  DEF shift(dx, dy) = yield void
                      EVOLVE Point(x0 + dx, y0 + dy)
}
```

Wenn wir diese „*Klasse*“ mit der ursprünglichen Form vergleichen, dann geben wir zum einen (implizit) die Signatur *PointSig* an. Andererseits übernehmen wir die JAVA-Konvention, dass die Klasse sowohl die Struktur als auch den Typ einführt. Was wir komplett weglassen, ist das „Lifting“ der Operationen auf die Versionen, die den Punkt als expliziten Parameter haben. Dieses Lifting erfolgt so mechanisch, dass es vom Compiler generiert werden kann.

Da dies kein Buch über Compilerbau ist, können wir den entsprechenden Übersetzungsprozess hier nicht im Detail analysieren; aber eine kleine Skizze soll zumindest andeuten, wie so etwas prinzipiell aussehen könnte. Abbildung 18.2 illustriert – in Anlehnung an entsprechende Konzepte aus UML– die wesentlichen Aspekte des Übergangs vom neuartigen Programm 18.2 zum klassischen Programm 18.1.

Der Begriff *Stereotype* besagt, dass ein neues syntaktisches Konstrukt – hier CLASS – eingeführt und auf bestehende Konzepte zurückgeführt wird. In unserem Fall haben solche Klassen einen Namen, eine Parameterliste, einen Typ (eine Signatur) und einen Rumpf. Die Klasse ist äquivalent zu einer Struktur gleichen Namens, in der auch ein Typ gleichen Namens eingeführt wird.

[2] Aber man darf nicht vergessen, dass es sich um die Simulation eines anderen Paradigmas im Rahmen der funktionalen Welt handelt. Solche Fremdkörper fügen sich meistens nur etwas sperrig in die Umgebung ein.

```
STEREOTYPE CLASS «Name»(«Parameter»): «Signature» = { «Body» }
IS
STRUCTURE «Name» = {
  TYPE «Name» = Obj «Signature»                                  -- Objekttyp
  FUN «Name»: «Parameter» → Beh «Name»                           -- Konstruktor
  DEF «Name»(«Parameter»)(t0) = (x, Δt0)
                                WHERE p(t0) = { «Body» }
  LIFT * «Body»
}
LIFT(FUN «op»: «α») IS (FUN «op»: «Name» → «α»)
LIFT(DEF «op»(«x») = «body») IS (DEF «op»(o)(«x»)(t) = (o(t).«op»(«x»))(t))
```

Abb. 18.2: Definition von Klassen als Strukturen

Dazu kommt dann noch ein Konstruktor. Außerdem muss aus dem Rumpf noch eine Sammlung von gelifteten Funktionen erzeugt werden.

Diese Skizze kann nur das prinzipielle Vorgehen andeuten. Man müsste noch genauer sagen, wie z. B. die «*Signature*», die in der Klasse implizit über die FUN-Klauseln angegeben ist, zum expliziten Typ gemacht werden kann. Außerdem muss man die Varianten für parameterlose Klassen beschreiben. Auch der unschöne Effekt, dass jetzt die Konstruktorfunktion (wie in JAVA) groß geschrieben wird, sollte noch repariert werden. Außerdem muss noch der Bezug der EVOLVE-Klausel auf das jeweils aktuelle Objekt formuliert werden. Diese und ähnliche Fragen führen in Bereiche des Compilerbaus, die heute unter dem Begriff *Reflection* intensiv vorangetrieben werden.

18.1.2 „Globale“ Objekte

In der praktischen Programmierung gibt es einen Effekt, der aus puristischer Sicht unvermeidlich ist, sich aber sehr störend auf die Lesbarkeit von Programmen auswirkt. Deshalb liegt es nahe, den Purismus zu opfern, um die Eleganz zu erhöhen.

Das Problem ist auch aus objektorientierten Programmen bekannt und zeigt sich z. B. deutlich am so genannten *Model-View-Control*-Paradigma (s. auch Kapitel 20). Hier hat man drei Objekte, die miteinander interagieren. Das würde man gerne folgendermaßen schreiben:

```
LET m ← model(v, c)    -- ------
    v ← view(m, c)     -- FALSCH!
    c ← control(m, v)  -- ------
IN  ...
```

Alle drei Objekte sollten einander kennen. Da sie aber erst zur Laufzeit generiert werden, muss man sie miteinander bekannt machen. Das geht jedoch

nicht in der oben beschriebenen Form, weil z. B. die Konstruktoroperation $model(v, c)$ zwei Objekte als Argumente hat, die es noch gar nicht gibt. Deshalb muss man zu relativ hässlichen Techniken greifen, mit denen man zuerst alle drei Objekte generiert und sie dann über spezielle Operationen miteinander bekannt macht. Das ist aber nicht das einzige Defizit. Auch wenn man die Objekte irgendwie miteinander bekannt gemacht hat, ist der jeweilige Bezug auf die Operationen der anderen nur relativ aufwendig und unleserlich programmierbar.

Dabei möchte man das Ganze doch nur so hinschreiben, wie das bei klassischen Strukturen aufgrund der Scopingregeln problemlos möglich ist:

OBJECT $Model = \{ \ldots View.paint(\ldots) \ldots \}$
OBJECT $View = \{ \ldots Model.access(\ldots) \ldots \}$
OBJECT $Control = \{ \ldots View.alert(\ldots) \ldots Model.set(\ldots) \ldots \}$

Eine ähnliche Situation findet sich z. B. im Compilerbau. Dort hat man Strukturen, die einen gegebenen Text gemäß einer vorgegebenen Grammatik analysieren. Diese Grammatik wird aber manchmal in einer Datei angegeben (z. B. bei DDTs in XML). Dann bietet sich folgende Struktur des Packages an:

STRUCTURE $Grammar \leftarrow readFile(\ldots)$
STRUCTURE $Compiler = MetaCompiler(Grammar)$

Da praktisch alle Funktionen im (Meta)Compiler auf die Grammatik Bezug nehmen, müssten sie normalerweise die Grammatik als expliziten Parameter mitschleppen. Durch die obige Konstruktion wird die Grammatik aber so zugreifbar, wie das bei gleichrangigen Strukturen in einer Gruppe üblich ist.

Dazu ist allerdings eine Konvention nötig, die eine leichte Verletzung des funktionalen Paradigmas darstellt. Zur Erinnerung: Die *IO*-Monade lässt sich nicht in umgebende, rein funktionale Ausdrücke einbetten. Genau das tun wir hier aber. Diese Verletzung grundlegender Prinzipien lässt sich durch eine Konvention mildern:

Festlegung (Initialisierung beim Programmstart)

Auf oberster Programmebene (also in Packages, Subpackages etc.) dürfen Objekte kreiert und Werte aus Dateien eingelesen werden. Eine adäquate Semantik dieser Konstruktionen wird durch folgende Compiler-Konvention sichergestellt: Die entsprechenden Programmelemente werden beim *Programmstart* initialisiert.

Der Compiler muss dabei sicherstellen, dass die Initialisierung keine zyklischen Abhängigkeiten erzeugt. (Die obige gegenseitige Bekanntschaft von *Model*, *View* und *Control* bedingt keine Abhängigkeit der Initialisierungsroutinen.) Außerdem dürfen nur lesende Zugriffe auf die Umgebung stattfinden, keine schreibenden.

18.2 Laufzeitsystem und andere Objekte (Zeit-Monaden)

Auf der Basis der Zeit-Monade lässt sich die Anbindung funktionaler Programme an das Laufzeitsystem wohl strukturiert und semantisch wohl definiert beschreiben. (Wir beschränken uns hier auf eine skizzenhafte Darstellung.)

Funktionale Sprachen wie OPAL oder HASKELL stellen große Bibliotheken von Funktionen für Ein-/Ausgabe und andere Systemdienste zur Verfügung. Tabelle 18.1 gibt einen kleinen Ausschnitt der notwendigen Strukturen an (und orientiert sich dabei an der BIBLIOTHECA OPALICA von OPAL, die in diesem Punkt etwas umfassender ist als die Standard Library von HASKELL).[3]

Struktur	Zweck
File	Zugriff auf Dateien
FileSystem	Zugriff auf das Dateisystem
Process	Elementare Prozesskontrolle des Betriebssystems
UserAndGroup	Zugriff auf die Benutzerverwaltung
Clock	Zugriff auf die Uhr des Betriebssystems

Tab. 18.1: Typische Strukturen eines Ein-/Ausgabe-Systems

Wir wollen im Folgenden zeigen, wie sich eine solche Bibliothek mit den hier erarbeiteten Konzepten sehr systematisch und wohl definiert konzipieren lässt. Die gesamte Laufzeitumgebung kann in ein Package mit entsprechenden Unterpackages organisiert werden. Das wird in Programm 18.3 angedeutet.

Programm 18.3 Das Package *Runtime* (Ausschnitt)

```
PACKAGE Runtime = {
  TYPE IO α = Beh α
  PACKAGE ProcessMgmt = {...}
  PACKAGE TimeMgmt = {...}
  PACKAGE UserMgmt = {...}
  PACKAGE FileMgmt = {...}
  ...
}
```

Wir greifen als Beispiel das Paket *FileMgmt* heraus. Es enthält als zentrale Strukturen die Verwaltung des gesamten Dateisystems sowie die Verarbeitung einzelner Dateien. Dazu kommen noch Hilfsstrukturen, in denen weitere

[3] Allerdings sind beide relativ klein, wenn man sie mit den Ein-/Ausgabe-Bibliotheken z. B. von JAVA oder dem .NET-System vergleicht. (Wir lassen hier offen, ob das gut oder schlecht ist.)

Aspekte des Dateimanagements definiert werden. Dies ist in Programm 18.4 skizziert.

Programm 18.4 Das Package *FileMgmt* (Ausschnitt)

```
PACKAGE FileMgmt = {
  OBJECT FileSystem = {...}        -- Verwaltung des Dateisystems
  CLASS File = {...}               -- Verwaltung einzelner Dateien
  STRUCTURE Path = {...}           -- Pfade
  STRUCTURE Properties = {...}     -- Eigenschaften von Dateien
  STRUCTURE Content = {...}        -- Inhalt von Dateien
  ...
}
```

Das ganze Dateisystem ist ein – riesiger – zeitabhängiger Wert und wird deshalb als OBJECT *FileSystem* eingeführt (vgl. Abschnitt 18.1.2). Auch die einzelnen Dateien sind zeitabhängige Werte und müssen daher als Objekte beschrieben werden. Aber im Gegensatz zum Dateisystem, das es nur einmal gibt, sind Dateien in großer Zahl vorhanden. Deshalb definieren wir für sie eine entsprechende Klasse.

Die anderen Aspekte des Packages sind klassische funktionale Strukturen. In der Struktur *Properties* wird alles zusammengefasst, was zu den Eigenschaften von Dateien gehört, also z. B. Zugriffsrechte, Erstellungs- und Änderungsdatum, assoziierte Icons etc. Die Struktur *Content* dient dazu, Dateiinhalte abstrakter zu erfassen. So kann man z. B. sequenzielle Dateien und Direct-Access-Dateien dadurch unterscheiden, dass *Content* α entweder *Seq* α oder *Array* α ist. In beiden Fällen muss aber noch das Konzept eines „Lesezeigers“ hinzukommen. (Details lassen wir hier offen; sie sind ohnehin mehr technischer als konzeptueller Natur.)

Das Objekt *FileSystem* enthält all diejenigen Operationen, die man zur Verwaltung von Dateien braucht. Ein kleiner Ausschnitt ist in Programm 18.5 gezeigt. Hier haben wir allerdings eine stark vereinfachte Sicht angegeben. Ein kleiner Blick auf die entsprechende Struktur in der BIBLIOTHECA OPALICA zeigt, dass in der Praxis ein Mehrfaches an Aufwand nötig ist, wenn man die gängigen Features von Betriebssystemen wie UNIX oder WINDOWS verfügbar machen will (z. B. Dateiart, Zugriffsrechte, Erstellungs- und Änderungszeit etc.).

Damit kommen wir auf der untersten Ebene zu den eigentlichen Dateien. Programm 18.6 zeigt einige typische Operationen der Klasse *File*. Wir haben die ungewöhnliche Entwurfsentscheidung getroffen, den Typ *File* polymorph zu machen. Das ist auch in den gängigen funktionalen Sprachen nicht üblich, obwohl es eigentlich naheliegend ist. Denn man sollte sich von der tra-

Programm 18.5 Das Objekt *FileSystem* (Ausschnitt)

```
OBJECT FileSystem = {
  FUN create: Path → Access → IO File     -- Erzeugen einer Datei
  FUN delete: File → IO Void              -- Löschen einer Datei
  FUN open: Path → Access → IO File       -- Öffnen einer Datei
  FUN close: File → IO Void               -- Schließen der Datei
  FUN link: File → Path → IO Void         -- zweiten Namen (als Link) setzen
  FUN rename: File → String → IO Void     -- Umbenennen
  FUN move: File → Path → IO File         -- Datei verschieben
  FUN copy: File → Path → IO File         -- Datei kopieren
  ...
}
```

Programm 18.6 Die Klasse *File* (Ausschnitt)

```
CLASS File = {
  TYPE File[α] = Obj(Content[α] × Properties)
  FUN stdIn stdOut stdErr: File
  FUN eof?: File → IO Bool                -- Dateiende erreicht?
  FUN length: File → IO Nat               -- Größe der Datei
  FUN name: File → IO String              -- Dateiname
  FUN path: File → IO Path                -- Dateipfad
  FUN read: File α → IO α                 -- nächstes Element lesen
  FUN read: File α → Nat → IO (Seq α)     -- die nächsten n Elemente lesen
  FUN readFile: File α → IO (Seq α)       -- ganze Datei lesen
  FUN skip: File α → Nat → IO Void        -- n Elemente überspringen
  FUN write: File α → α → IO Void         -- Element schreiben
  FUN flush: File α → IO Void             -- Puffer auf Datei schreiben
  FUN rewind: File α → IO Void            -- Zurücksetzen an Anfang
  FUN read: File Char → Nat → IO String   -- die nächsten n Zeichen lesen
  FUN readLines: File Char → IO(Seq String) -- restliche Zeilen lesen
  FUN write: File Char → String → IO Void -- String schreiben
  FUN writeLn: File Char → String → IO Void -- String und Newline schreiben
  FUN writeLines: File Char → Seq String → IO Void -- Zeilen schreiben
  ...
}
```

ditionellen, rein technisch orientierten Sichtweise lösen und einen abstrakteren Zugang zu Dateien suchen. In den alten maschinennahen Sprachen wie C war es akzeptabel, eine Datei im Wesentlichen als Byte-Array aufzufassen. Aber in funktionalen Sprachen sollte man den Inhalt typisiert auffassen; die entspre-

chenden Konvertierungsfunktionen einzufügen, ist die Aufgabe des Compilers, nicht des Programmierers.

Allerdings haben wir es uns in Programm 18.6 ein bisschen leicht gemacht, denn der polymorphe Typ α müsste noch durch eine geeignete Typklasse eingeschränkt werden.

Anmerkung: Die dafür notwendigen Techniken sind spätestens durch JAVA *bekannt geworden. Dort kann man mittels Serialization nahezu beliebige Werte auf Dateien schreiben und wieder einlesen. Das* JAVA*-Interface* `Serializable` *entspricht also im Wesentlichen der Typklasse, die wir als Einschränkung bei File* α *fordern müssen. In* JAVA *sieht man übrigens auch erste schüchterne Versuche in Richtung auf typisierte Dateien: Die Unterscheidung in Byte-orientierte* `Stream`*-Dateien und* UNICODE*-orientierte* `Reader` *und* `Writer` *stellt eine erste rudimentäre Form von Typisierung dar.*

In der Praxis hat man in der Struktur *File* noch eine Fülle von weiteren Operationen, mit denen man insbesondere die verschiedenen Attribute abfragen und setzen kann. (Da auch dieser Aspekt wieder mehr technischer als konzeptueller Natur ist, betrachten wir ihn hier nicht weiter.)

18.2.1 Dateioperationen höherer Ordnung

Interessanter ist ein weiterer Aspekt, der im Rahmen von funktionaler Programmierung eigentlich auch bei Ein-/Ausgabe erwartet werden muss. Funktionen höherer Ordnung, insbesondere das *Map-Filter-Reduce*-Paradigma, sollten auch für Dateien verfügbar sein. In Programm 18.7 sind die entsprechenden Operationen angegeben.

Programm 18.7 Die Struktur *HigherOrderFile* (Ausschnitt)

```
CLASS HigherOrderFile = {
  FUN _*: (α → β) → File α → IO (File β)      -- Map
  FUN _◁: (α → Bool) → File α → IO (File α)   -- Filter
  FUN _/: (α × α → α) → File α → IO α         -- Reduce
  ...
}
```

Diese Operationen orientieren sich an den entsprechenden Funktionalen auf Sequenzen, die in Abschnitt 1.2 angegeben sind. (Die dort aufgeführten Varianten ließen sich auch hier hinzufügen.) Wir verzichten darauf, die – offensichtlichen – Implementierungen anzugeben. Als einzige Besonderheit sollte erwähnt werden, dass bei *Map* und *Filter* jeweils eine neue (anonyme) Datei kreiert wird, in die die Daten hineingeschrieben werden. Diese wird als Resultat zurückgeliefert und muss bei Bedarf noch (mittels *rename*) einen externen Namen erhalten.

18.2.2 Ein typisiertes Dateisystem?

In Betriebssystemen werden Dateien weitgehend uniform behandelt, obwohl es zahlreiche Typaspekte gibt: Man unterscheidet z. B. normale Dateien, Pipes, Directorys, symbolische Links etc. Ein schüchterner Versuch zur Typisierung erfolgt über normierte Dateiendungen wie `.txt`, `.exe`, `.jpg` usw. Etwas weiter gehen Versuche, mit so genannten MIME-Kennungen zu arbeiten. Außerdem lassen sich Dateien zum Lesen, Schreiben oder Ausführen öffnen.

Auch wenn man sich JAVA oder das .NET-System ansieht, entdeckt man eine reichhaltige Typisierung von Ein-/Ausgabe-Klassen (die sich in einer relativ tiefen Vererbungshierarchie widerspiegelt). Allerdings erkennt man an dem völlig überfrachteten und nur technisch motivierten Zusammenspiel von Klassen wie `InputStream`, `FileInputStream`, `FilterInputStream`, `BufferedInputStream`, `Reader`, `FileReader`, `BufferedReader` usw. auch, dass es ein nichttriviales Problem ist, hier eine adäquate Typisierung zu entwickeln. (Vor allem sieht man aber, wie wichtig es ist, technischen Ballast vom Compiler erledigen zu lassen und nicht vom Programmierer.)

Wenn man in dieser Richtung weiter denkt, ergeben sich Subtyp-Relationen der folgenden Art:

```
TYPE Dir ⊆ File          -- (problematisch)
TYPE Link α ⊆ File α
TYPE Pipe α ⊆ File α
  ...
```

Allerdings sieht man hier, dass bei *Dir* die passende Angabe der Generizität etwas Probleme macht: Man bräuchte Polymorphie höherer Ordnung [117]. Als Alternative könnte man auch Directorys und normale Dateien als völlig verschiedene Dinge auffassen, die Varianten eines gemeinsamen Summentyps sind. Für *Directorys* hat man ohnehin ganz andere Operationen als für normale Dateien. Insbesondere möchte man häufig Auflistungen der enthaltenen Dateien haben, was auf spezielle Varianten des *Map-Filter-Reduce*-Paradigmas hinausläuft.

Trotz dieser Schwierigkeiten wäre es gerade bei funktionalen Sprachen angemessen, ein solches Typsystem zu entwickeln. Dies muss allerdings heute noch als offener Forschungsauftrag gelten.

19

Agenten und Prozesse

Viele Köche verderben den Brei.
(Sprichwort)

In vielen Situationen muss man die Arbeit verteilen, so dass Teilaufgaben simultan erledigt werden. Dabei kann „simultan“ heißen, dass die verschiedenen Aktivitäten tatsächlich gleichzeitig ablaufen; wir sprechen dann von *parallelen Prozessen*. Aber in der Praxis genügt es oft, mit so genannten *pseudo-parallelen Prozessen* zu arbeiten. Das bedeutet, dass jeder einzelne Prozess in kleine Zeitscheiben zerlegt wird und dass diese Fragmente miteinander verschränkt (engl.: *interleaved*) abgearbeitet werden. Bei solchen pseudo-parallelen Prozessen finden also alle Arbeiten nach wie vor sequenziell statt. Aber wegen der immensen Geschwindigkeit heutiger Prozessoren wirkt das für die Umgebung – insbesondere für menschliche Nutzer – als ob alle Prozesse gleichzeitig ablaufen würden.

Es gibt zwei grundsätzlich verschiedene Ziele für die Einführung von Parallelverarbeitung. Die eine Variante dient dazu, Berechnungen zu *beschleunigen*, indem man sie in Teilberechnungen zerlegt und auf mehrere Rechner verteilt. Dazu braucht man also echte Parallelverarbeitung auf mehreren Prozessoren. Die andere Variante wird benötigt, wenn man es mit Prozessen zu tun hat, die inhärent nebenläufig sind und miteinander *kooperieren*. Hier genügt häufig eine pseudo-parallele Verarbeitung auf einem Prozessor. Diese Situation betrachten wir im Folgenden.

Ein typisches Szenario liegt bei Programmen mit interaktiven graphischen Benutzerschnittstellen, kurz *GUIs*, vor (s. Kapitel 20). Dem Benutzer stehen mehrere Fensterelemente mit unterschiedlichen Optionen zur Verfügung: Editieren in Textfenstern, Klicken auf Buttons, Verschieben von Scrollbars usw. Auf jede dieser möglichen Benutzeraktivitäten muss das Programm entsprechend reagieren. Da völlig unvorhersehbar ist, wann der Benutzer was tun wird, muss das Programm simultan auf alles vorbereitet sein.

Als beste Lösung für diese Art von Szenarien hat sich herauskristallisiert, das Programm in Form von (pseudo-)parallel laufenden Einheiten zu organi-

sieren, die sich jeweils um eine der möglichen Benutzeraktivitäten kümmern. Man spricht dann von kooperierenden Systemen. Falls die einzelnen Prozesse auf unterschiedlichen Rechnern laufen, spricht man auch von *verteilten Systemen*. Zur programmiersprachlichen Repräsentation solcher Prozesse werden wir *Agenten* einführen.

Es gibt in der Informatik verschiedene Ansätze, um die Interaktion zwischen den Prozessen eines kooperierenden Systems zu realisieren. (Eine nach wie vor lesenswerte Übersicht findet sich in [74]). Die wohl älteste Technik ist die der *gemeinsamen Variablen* (engl.: *shared variables*). Hier lesen und schreiben mehrere Prozesse die gleichen Variablen. Damit dies konfliktfrei funktioniert, müssen Techniken zur Synchronisation des Zugriffs eingeführt werden (Locks, Semaphore, Monitore etc.). Die heute wohl bekannteste Realisierung dieser Technik findet sich in den *Threads* von JAVA.

Eine andere naheliegende Idee sind (synchrone oder asynchrone) *Kanäle*. Über diese Kanäle fließen Datenströme zwischen den Prozessen. Diese Technik ist aus Sicht guten Software-Engineerings besser als die der gemeinsamen Variablen, weil sie robuster und weniger fehlerträchtig ist. (Das ist auch der Grund, weshalb viel Kritik an den JAVA-Threads geäußert wird.) Allgemeiner als Kanäle sind jedoch die so genannten *Service-Access-Points*, kurz: *SAPs*; deshalb ziehen wir diese im Folgenden als Grundlage unseres Designs heran.

Die Integration von Parallelität in funktionale Sprachen ist ein relativ junger Forschungsgegenstand und daher stark im Fluss. Am Ende dieses Kapitels gehen wir deshalb kurz auf einige der aktuell diskutierten Sprachentwürfe ein.

19.1 Service-orientierte Architekturen

Im Bereich des Software-Engineerings kooperierender und verteilter Systeme haben sich als zentrale Ideen Konzepte wie Agenten und Client-Server-Architekturen etabliert. Neuerdings ist als weiteres beliebtes Schlagwort der Begriff „Service-orientierte Architekturen“, kurz *SOA*, hinzugekommen. Die entsprechenden Prinzipien erleichtern nicht nur die Organisation großer Software-Systeme, sondern versprechen auch beim Design von individuellen Programmen einen Nutzen.

Erfreulicherweise treffen die Prinzipien des Service-orientierten Designs ziemlich genau die Konzeption, die z. B. in der Sprache OPAL schon in den 90er Jahren für die Implementierung kooperierender Prozesse herangezogen wurde. Deshalb werden wir uns im Folgenden an diesen Ideen orientieren, wobei wir allerdings versuchen werden, sie auf der Basis unserer Sprachmittel etwas abstrakter und eleganter zu fassen. Im Gegensatz zum Gebrauch im Software-Engineering, wo der Begriff der Service-Orientierung meistens recht allgemein und unscharf bleibt, werden wir das Konzept hier sehr genau und technisch präzise fassen.

Abbildung 19.1 illustriert ein Service-orientiertes Design. Dabei haben wir eine an UML angelehnte Darstellungsform gewählt. Das kooperierende System

besteht aus *Komponenten*, die wir als *Agenten* bezeichnen, und aus *Konnektoren*, die wir als *Service-Access-Points*, kurz *SAPs*, bezeichnen.

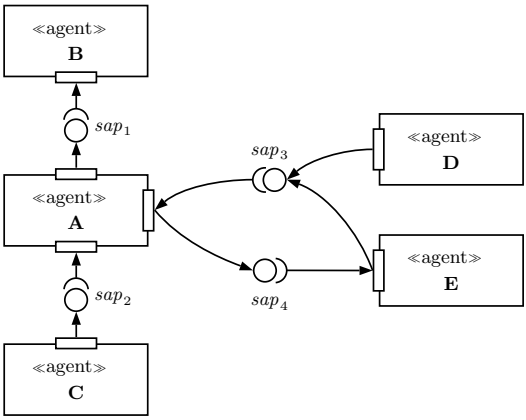


Abb. 19.1: Service-orientierte Architektur

Bevor wir (in den folgenden Abschnitten 19.2 und 19.3) im Detail auf die programmiersprachliche Realisierung von Agenten und SAPs eingehen, wollen wir erst noch die Grundidee ihres Zusammenspiels skizzieren.

Definition (Service-orientiertes System)

Ein *Service-orientiertes System* besteht aus Agenten und SAPs:

- Die *Agenten* sind die aktiven Einheiten im System.
- Agenten können *Dienste* nachfragen und anbieten.
- Angebot und Nachfrage von Diensten werden an *Service-Access-Points (SAPs)* koordiniert.

Man bezeichnet diejenigen Agenten, die eine Nachfrage nach Diensten stellen, als *Clients* und diejenigen Agenten, die die Dienste anbieten und erbringen, als *Server*. Dieses *Client-Server-Modell* besitzt einen hohen Grad an Flexibilität, wie man an Abbildung 19.1 erkennen kann.

- Es können mehr als zwei Agenten über einen SAP miteinander interagieren. So kommunizieren in Abbildung 19.1 an sap_3 die drei Agenten A, D und E miteinander.
- Die Rollen von Clients und Servern sind nicht fest zugeordnet. Derselbe Agent kann in einer Interaktion als Client und in einer anderen als Server auftreten. Oft muss sogar ein Server, der einen Dienst erbringen will, zu diesem Zweck bei einem anderen Agenten als Client auftreten. In Abbildung 19.1 fungiert der Agent A am sap_1 als Server und am sap_2 als Client.

Bei der Interaktion zwischen A und E hat sogar jeder der beiden Agenten beide Rollen. (Vorsicht: Solche Situationen können leicht zu Deadlocks führen!)

- Ein besonders wichtiger Aspekt ist in Abbildung 19.1 nicht zu sehen: Über SAPs können nicht nur Daten fließen, sondern komplexe Dienste angefordert und ihre Ergebnisse zurückgeliefert werden. Dies werden wir in Abschnitt 19.3 im Detail studieren.

In den bildlichen Darstellungen (im Stil von Abbildung 19.1) werden wir weitestgehend die Konvention einhalten, dass der volle Kreis für das Dienstangebot auf der Serverseite steht und der Halbkreis für die Dienstnachfrage auf der Clientseite; es wird aber Situationen geben, in denen sich diese visuelle Hilfe nicht durchhalten lässt.

19.2 Agenten als Monaden

Als ersten zentralen Bestandteil unseres Service-orientierten Designs betrachten wir die aktiven Einheiten, also die Agenten.

Definition (Agent)

Ein ***Agent*** ist durch folgende Eigenschaften charakterisiert:

- Er ist eine autonome Programmeinheit.
- Er führt ein sequenzielles Programm – eine monadische Operation – aus.
- Er arbeitet (pseudo-)parallel zu anderen Agenten.
- Er kann bei seiner Terminierung ein Ergebnis abliefern.

Der letzte Punkt hat Vor- und Nachteile. Auf der einen Seite fügt sich das Abliefern von Ergebnissen gut in die funktionale Welt ein. Auf der anderen Seite entsteht damit aber ein kritisches Programmkonstrukt: Ausdrücke mit Seiteneffekt.

Diese Definition korrespondiert zum Verhalten von Monaden, mit denen wir Agenten daher auch hervorragend darstellen können. Die Struktur *Agent* in Programm 19.1 stellt Funktionen zur Kreation und Behandlung von nebenläufigen Agenten bereit. Sie basiert auf der Zeit-Monade von Programm 17.3; alle Operationen von Agenten sind vom Typ *Beh* α.

Die Operation *agent* kreiert einen neuen Agenten. (In HASKELL heißt die entsprechende Operation *forkIO*.) Die Operation *agent*(*op*) erhält als Argument eine (monadische) Operation, die das „Programm“ des Agenten beschreibt. Diese Operation wird unmittelbar nach der Kreierung des Agenten als eigenständiger Prozess (Thread) gestartet und läuft von da an (pseudo-)parallel zu dem Prozess ab, der gerade die Operation *agent*(*op*) ausgeführt hat. Das Ergebnis von *agent*(*op*) ist der neu kreierte Agent. Damit werden z. B. Anwendungen folgender Art möglich:

Programm 19.1 Die Struktur *Agent*

```
STRUCTURE Agent = {
  TYPE Agent[α] = «interne Repräsentation von Agenten»
  FUN agent: Beh α → Beh(Agent α)
  DEF agent(op) = «erzeuge und starte Agent mit Verhalten op»
  FUN await: Agent α → Beh α
  DEF await(a) = «warte auf die Terminierung des Agenten a»
  FUN kill: Agent α → Beh Void
  DEF kill(a) = «veranlasse die Terminierung des Agenten a»
  FUN self: Beh(Agent α)
  DEF self = «die Identität des Agenten selbst»
}
```

$$\ldots \,\&\, agent(code) \to a \,\&\, \ldots \,\&\, await(a) \to x \,\&\, \ldots$$

Hier wird ein Agent kreiert, auf den man sich im weiteren Programmverlauf unter dem Namen a beziehen kann. Dieser Agent führt seine Operation *code* parallel zu den weiteren Aktivitäten „ . . . “ des aktuellen Prozesses aus. Dann wartet der aktuelle Prozess auf die Terminierung des Agenten a. Dabei wird das Ergebnis der Operation *code* an den Namen x gebunden.

Normalerweise endet das „Leben“ eines Agenten, wenn er sein Programm fertig abgearbeitet hat. Es ist aber auch möglich, Agenten mit roher Gewalt abzubrechen. Dazu dient die Operation *kill*.

Anmerkung: Die Operation kill ist implementierungstechnisch mit Vorsicht zu genießen, wie man z. B. aus der (sehr wechselhaften) Geschichte der entsprechenden Operationen in JAVA *ablesen kann. Das Problem ist, dass ein Prozess nicht abrupt „abgeschossen“ werden darf, wenn er gerade eine kritische Phase durchläuft. Man muss ihm eine Chance geben, noch „ordentlich aufzuräumen“. Das hat in* JAVA *im Laufe der Releases zu entsprechenden Redefinitionen geführt.*

Eine typische Anwendung der Operation *self* besteht darin, dass ein Agent a einem anderen Agenten b mitteilt, wer er ist. (Dies geschieht über eine entsprechende Interaktion an einem SAP.) Dann kann b z. B. auf a warten oder – im Extremfall – auch a „killen“.

Das folgende Beispiel illustriert eine einfache aber nützliche Anwendung von Agenten: Man kann für Berechnungen Zeitbeschränkungen vorgeben.

Beispiel 19.1 (*watchdog*)

Die Funktion *watchdog* überwacht die Ausführung eines (monadischen) Programmstücks, um es abbrechen zu können, wenn ein vorgegebenes Zeitlimit überschritten ist. Dazu benötigen wir die Operationen _ + _, *timeout* und *fail* aus der Struktur *TimeMonad* von Programm 17.3 auf Seite 379.

```
FUN watchdog: Beh α × Nat → Beh α
DEF watchdog(op, time) =
    LET
        worker ← agent(op)
    IN
    await(worker)
    +
    timeout(time) & kill(worker) & fail("timed out")
```

Man beachte, dass der Typ *Beh* α implizit auf dem Typ *Maybe* α basiert. Deshalb kann auch mittels *fail* eine Exception geliefert werden.

In diesem Beispiel gibt es bei der Auswahl zwei Möglichkeiten:

- Entweder der Agent *worker* wird mit der Berechnung seines Programms *op* rechtzeitig fertig; dann „gewinnt“ der Zweig *await*(*worker*) und die gesamte Auswahl und damit auch die Funktion *watchdog* endet mit dem Ergebnis von *op*.
- Oder der Agent braucht zu lange; dann „gewinnt“ der Zweig mit *timeout* und der Agent *worker* wird abgebrochen. Dann endet die gesamte Auswahl und damit auch die Funktion *watchdog* mit der Exception.

Aber es gibt eine schwerwiegende Komplikation: Damit das obige Programm so funktioniert, wie wir das erwarten, muss der Compiler ***preemtive Multitasking*** implementieren. Das heißt, zwischen den (pseudo-)parallelen Prozessen muss in hinreichend kurzen Intervallen ein Wechsel stattfinden, so dass es auch eine „zeitnahe“ Chance des Unterbrechens gibt. Voraussetzung dazu ist, dass auch das zugrunde liegende Betriebssystem diese Eigenschaft hat.

Dieses *preemtive Multitasking* ist nicht in allen Sprachen gegeben. So ist z. B. OPAL nicht *preemptive*, weil Prozesswechsel immer nur bei den monadischen Operatoren möglich sind. Wenn in einer solchen Sprache in der Operation *op* des obigen Agenten *worker* eine sehr lange, rein funktionale Berechnung stattfindet, dann kann sie nicht durch das *timeout* abgebrochen werden.

19.3 Kommunikation: Service-Access-Points

Wie schon eingangs dieses Kapitels diskutiert, ist die eleganteste Möglichkeit der Interaktion von Agenten das Konzept der so genannten *Service-Access-Points*. Diese schließen als Spezialfall das bekanntere Konzept der *Kanäle* mit ein.

Definition (Service-Access-Point (SAP))

Im Client-Server-Modell kommunizieren die Agenten über *Service-Access-Points*, kurz: *SAPs* (vgl. Abbildung 19.1). Diese Interaktion geschieht nach den folgenden Prinzipien:

- Jeder Agent, der einen SAP kennt, kann dort Dienste anbieten und/oder nachfragen.
- Das Anbieten und Nachfragen sind monadische Operationen.
- Die tatsächliche Erledigung eines Dienstes findet in Form eines *Rendezvous* statt: Ein passendes Angebot-Nachfrage-Paar führt dazu, dass die zugehörige Operation ausgeführt wird. Die Ausführung obliegt dem Server; der Client „ruht“ inzwischen.
- Am Ende des Rendezvous besitzen sowohl Client als auch Server das Ergebnis des Dienstes und fahren (parallel) mit ihrer jeweiligen Arbeit fort.

Die technischen Details dieser Prinzipien werden im Folgenden genauer ausgearbeitet. Dabei gibt es zwei Formen der programmiertechnischen Realisierung.

- Man kann die Angebote und Nachfragen als *Datentypen* kodieren, üblicherweise in Form von entsprechenden Summentypen. Diese Variante ist z. B. in OPAL implementiert.
- Man kann aber auch Angebote und Nachfragen direkt als Operationen auffassen, die dann in geeigneten *Strukturen* definiert werden müssen. Dieser Ansatz ist abstrakter und eleganter, braucht aber einige der fortgeschrittenen Sprachmittel, die wir in den früheren Kapiteln eingeführt haben.

Wir zeigen hier die modernere Variante, die Konzepte wie Typklassen verwendet; die traditionelle Variante wird kurz in Abschnitt 19.7 angesprochen. Programm 19.2 enthält die Struktur *Service*, die die Typklasse *Sap* der Service-Access-Points definiert und die Operationen zur Service-Nachfrage und zum Service-Angebot einführt. Diese Struktur – die ja nur einmal für die Bibliothek vordefiniert wird – ist so gestaltet, dass die *Verwendung* von SAPs in Anwendungsprogrammen möglichst einfach und intuitiv verständlich erfolgen kann.

Leider ist unser Typsystem zu schwach, um eine essenzielle Zusatzbedingung auszudrücken:[1] Bei Angebot und Nachfrage müssen für die entsprechenden Dienste – also die Funktionen vom Typ $(\alpha \to Beh\ \beta)$ etc. – im Sap-Typ σ geeignete Gegenstücke vorhanden sein; das wird in einem Beispiel gleich deutlicher illustriert werden. Zuvor wollen wir jedoch die einzelnen Funktionen näher erläutern.

Dienstnachfrage durch den Client

An einem SAP kann ein Client eine Dienstnachfrage *service*(*request*) anmelden; ein solcher Dienst ist grundsätzlich eine monadische Operation. Ein Server kann diesen Dienst erbringen und die Antwort über den SAP kommunizieren. Diese Antwort wird der Continuation des Clients übergeben. Wir

[1] Grundsätzlich wäre dies schon machbar, aber der Aufwand ist sehr hoch. Wir müssten die Typklasse *Sap* als die Menge aller Gruppentypen charakterisieren, deren Komponenten ausschließlich „Dienste“ sind, also Funktionen mit Ergebnistyp $Beh\ \alpha$.

Programm 19.2 Die Struktur *Service* der Service-Access-Points

```
STRUCTURE Service = {
  TYPECLASS Sap
  FUN sap: σ ↦ Beh σ   VAR σ: Sap
  DEF sap = «erzeuge neuen Service Access Point»
  FUN _._ _: σ: Sap × (α → Beh β) × α → Beh β
  DEF sap.service(request) = «Nachfrage eines Dienstes am SAP»
  FUN _ ⋉ _: σ: Sap × (α → Beh β) → Beh β
  FUN _ ⋉ _: σ: Sap × (α → Beh (β × γ)) → Beh γ
  DEF sap ⋉ service = «Angebot eines Dienstes am SAP»
  FUN _ ⋉ _ | _: σ: Sap × (α → Beh β) × (α → Bool) → Beh β
  FUN _ ⋉ _ | _: σ: Sap × (α → Beh (β × γ)) × (α → Bool) → Beh γ
  DEF (sap ⋉ service | cond) = «Angebot eines bedingten Dienstes am SAP»
}
```

haben also üblicherweise Programmstücke der folgenden Bauart, wobei (im einfachen Standardfall) *service* den Typ $(\alpha \to Beh\ \beta)$ hat, *request* den Typ α und somit *answer* den Typ β.

$$\ldots (sap.service(request)) \to answer\ \&\ continuation \ldots$$

Man beachte, dass die Punktnotation so gewählt ist, dass sich der Operationsaufruf an einem SAP genauso liest wie z. B. der Aufruf einer Operation eines Objekts. Damit werden die Client-Programme durch die SAP-Kommunikation notationell nicht belastet; sie lesen sich nahezu funktional, höchstens „normal“ monadisch.

Zu dieser (vorgespiegelten) notationellen Vertrautheit trägt auch ganz wesentlich der Trick bei, die Operation _._ _ mit *drei* Argumenten zu versehen, so dass die beiden letzten sich optisch wie die zugehörige Funktionsapplikation präsentieren. (Diese Applikation findet aber tatsächlich erst auf der Server-Seite statt.)

Dienstbereitstellung durch den Server

Ein Server kann an einem SAP einen Dienst *service* anbieten. Wenn eine Nachfrage von einem Client vorliegt, führt der Server seinen Dienst aus und legt das Resultat am SAP ab, das der Client dann dort übernehmen kann. Als Besonderheit kann auch der Server selbst mit dem Resultat in seiner Continuation weiterarbeiten. Auf Serverseite haben wir also (im einfachen Standardfall) Programmstücke folgender Bauart.

$$\ldots (sap \bowtie service) \to answer\ \&\ continuation \ldots$$

Es gibt auch eine bedingte Variante des Dienstangebots.

$$\ldots (sap \bowtie service \mid cond) \to answer\ \&\ continuation \ldots$$

Hier wird der Dienst nur ausgeführt, wenn die Nachfrage(daten) die Bedingung *cond* erfüllen.

Natürlich kann der Server nur solche Dienste anbieten, die er auch implementiert hat. Das heißt, er muss eine Definition der folgenden Bauart enthalten:

FUN $service$: $\alpha \to Beh\ \beta$
DEF $service(req) = \ldots$

Man beachte, dass hier bzgl. der Typkorrektheit eine Besonderheit gilt. Bei Diensten gehört der *Name* essenziell zum Typ. Es kann also nicht irgendeine Funktion angeboten werden, die den Typ $(\alpha \to Beh\ \beta)$ hat. (Insofern sind Dienste analog zu Konstruktorfunktionen in Produkt- und Summentypen.)

Dienstkoordination durch ein Rendezvous

Das Dienstangebot des Servers und die Dienstnachfrage des Clients werden durch ein *Rendezvous* miteinander koordiniert. Der konkrete Ablauf dieses Rendezvous wird durch das *Sequence-Chart* in Abbildung 19.2 beschrieben.

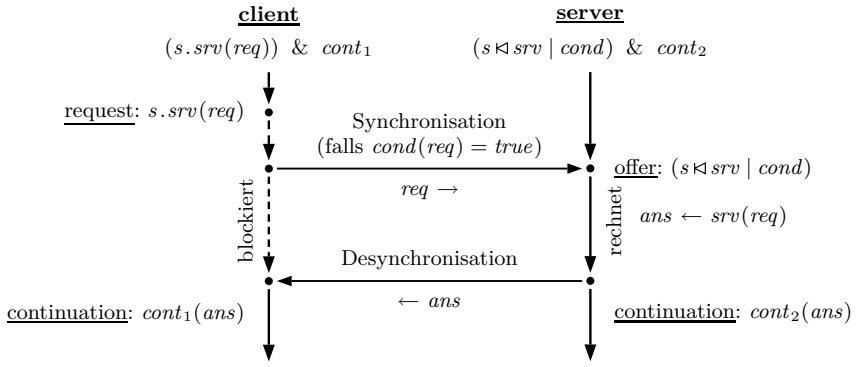


Abb. 19.2: Ablauf eines Rendezvous

Wenn der Client seine Anfrage $(s.srv(req))\ \&\ cont_1$ stellt, wird der Server im Allgemeinen nicht bereit sein. Deshalb wird der Client blockiert, bis der Server ein (passendes) Angebot $(s \bowtie srv \mid cond)\ \&\ cont_2$ macht. In diesem Augenblick findet das Rendezvous statt.

Zunächst wird geprüft, ob die Anfrage *req* die im Serviceangebot geforderte Bedingung $cond(req)$ erfüllt. Ist das nicht der Fall, so wird das Rendezvous abgebrochen und weiter gewartet. Anderenfalls wird der Wert *req* vom Server übernommen, der damit die Funktion $srv(req)$ auswertet. (Der Client muss weiter warten.) Das Ergebnis *ans* der Berechnung wird schließlich an den

Client zurückgeschickt, der jetzt weiterarbeiten kann und seine Continuation $cont_1$ auf *ans* anwendet. Auch der Server rechnet mit seiner Continuation $cont_2$ weiter, die er ebenfalls auf *ans* anwendet. (In der Praxis wird er allerdings *ans* meistens ignorieren.)

Es gibt natürlich auch den dualen Fall, bei dem der Server auf den Client wartet.

Für die Konkurrenz mehrerer Server bzw. Clients an einem SAP treffen wir die folgende Festlegung. Sie ist zwar nicht unstrittig, weil sie implementierungstechnisch etwas mehr Aufwand verursacht, aber viele Programme werden damit erheblich vereinfacht – und das ist unsere primäre Motivation.

Festlegung (SAPs sind fair)

Wenn an einem SAP mehrere Server den gleichen Dienst anbieten, werden sie in der Reihenfolge berücksichtigt, in der sie sich angemeldet haben. Das Entsprechende gilt für die Nachfragen von Clients.

Eine Verallgemeinerung

In der Praxis gibt es Situationen, für die erweiterte Formen der Typisierung benötigt werden. Diese Varianten betreffen alle den Typ des Dienstes *service*, weil hier beim Server komplexere Situationen vorliegen.[2]

- Als Erstes kann es vorkommen, dass der Server einen anderen Wert für die Continuation braucht als der Client. Dann berechnet der Service beide Werte und der Client benutzt den ersten, der Service selbst den zweiten.

  FUN $service$: $\alpha \to Beh\ (\beta \times \gamma)$
  DEF $service(req) = \ldots yield(b, c)$

  Man beachte, dass wegen der Nichtassoziativität der Produktbildung die Typen β und γ selbst wieder Tupel sein können, so dass die jeweiligen Continuations auch mit mehreren Werten versorgt werden können.
  Der eingangs behandelte einfache Typ $(\alpha \to Beh\ \beta)$ ist offensichtlich nur eine Abkürzung für den allgemeinen Typ in der speziellen Situation $(\alpha \to Beh(\beta \times \beta))$ mit $yield(b, b)$.
- Außerdem kann es vorkommen, dass der Server zur Diensterbringung weitere Parameter braucht, die der Client gar nicht kennt. Deshalb gibt es auch Dienste folgender Form:

[2] Diese Erweiterungen beruhen auf Erfahrungen mit OPAL. Hier treten immer wieder bestimmte Situationen auf, die auf der Anwendungsseite eine relativ aufwendige und unleserliche Programmierung erfordern [53, 54]. Diese Situationen sind im hier beschriebenen erweiterten Konzept auf der Bibliotheksseite eingebaut.

FUN $service\colon\ \delta \to (\alpha \to Beh\ \beta)$
FUN $service\colon\ \delta \to (\alpha \to Beh\ (\beta \times \gamma))$
DEF $service(data)(req) = \ldots$

Diese Dienste werden dann in der folgenden Form angeboten:

$\ldots\, sap \bowtie service(d) \ldots$

Wir haben mit diesem Design die Situation für den Programmierer optimiert und müssen deshalb hohe Anforderungen an die Fähigkeiten des Compilers stellen:

- In den Signaturen der SAP-Typen wird nur die *für den Client sichtbare Funktionalität* der Dienste angegeben. Dies respektiert das fundamentale Hiding-Prinzip des Software-Engineerings.
- Beim Server prüft der Compiler nach, ob die gegebene Implementierung mit dieser Funktionalität „kompatibel" ist. Die zulässigen Möglichkeiten sind in Tabelle 19.1 zusammengefasst.

Service-Typ im Sap*:* TYPE $S\colon \mathit{Sap} = \{\ldots, service\colon\ \alpha \to Beh\ \beta, \ldots\}$	
Möglichkeiten im Server:	
Implementierung	*Angebot*
FUN $service\colon\ (\alpha \to Beh\ \beta)$	$\ldots\, sap \bowtie service \ldots$
FUN $service\colon\ \delta \to (\alpha \to Beh\ \beta)$	$\ldots\, sap \bowtie service(d) \ldots$
FUN $service\colon\ (\alpha \to Beh(\beta \times \gamma))$	$\ldots\, sap \bowtie service \ldots$
FUN $service\colon\ \delta \to (\alpha \to Beh(\beta \times \gamma))$	$\ldots\, sap \bowtie service(d) \ldots$

Tab. 19.1: Variationen für Diensttypen

19.4 Ein Beispiel

Zur Illustration der Agenten-Interaktion über SAPs betrachten wir ein kleines, aber typisches Beispiel, bei dem vier Arten von Agenten über insgesamt vier SAPs miteinander koordiniert werden.

Beispiel 19.2 (Beschränkter Informationszugriff)

Wir betrachten das Szenario von Abbildung 19.3. Es gibt eine beliebige Anzahl von Clients, die Informationen wollen, die von einem oder mehreren Repository-Servern bereitgestellt werden. Aber nicht jeder Client darf auf alle Informationen zugreifen. (Man kann z. B. an die Telefonauskunft einer Firma denken, wo auch nicht jedermann alle internen Telefonnummern erfahren darf.) Deshalb gehen die Anfragen nicht direkt an die Repository-Server,

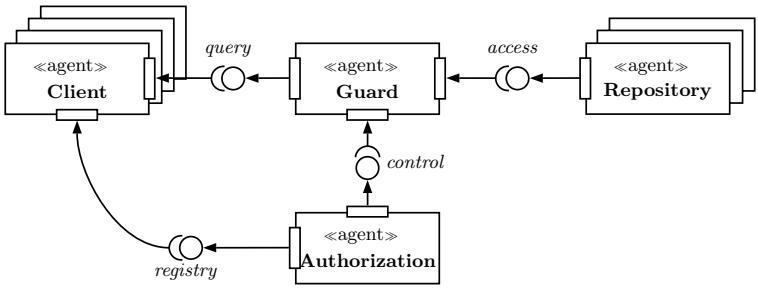


Abb. 19.3: Service-orientierte Architektur

sondern werden von einem Guard weitergeleitet, der vorher die Berechtigung prüft.

Die Berechtigungen werden von einem Authorization-Server verwaltet, bei dem sich jeder Client anmelden muss, bevor er Anfragen an das Repository stellt.

Der gesamte Ablauf verwendet eine Reihe von Hilfstypen, auf die wir hier nicht näher eingehen: *Question*, *Answer*, *Identification*, *Challenge*, *Response* und *Certificate*.

Die SAPs

SAPs repräsentieren im Wesentlichen Signaturen, genauer: Sammlungen von Funktionstypen. Diese Funktionen müssen die Zusatzbedingung erfüllen, dass ihr Resultattyp jeweils *Beh* ... ist. Das ist das primäre Charakterisitikum der Typklasse `Sap`.

Programm 19.3 enthält die Typen der SAPs aus Abbildung 19.3. Wie man sieht, ist jeder konkrete SAP-Typ eine Signatur. Diese Signatur umfasst diejenigen Dienste, die an dem SAP zwischen Clients und Servern ausgetauscht werden können.

Programm 19.3 enthält nur die Typen der SAPs. Die konkreten SAPs selbst müssen aber noch generiert werden. Deshalb stehen in einem Programmsystem, das das Szenario aus Abbildung 19.3 realisiert, üblicherweise die folgenden Konstruktoren:

```
...
LET query: Query ← sap[Query]
    registry: Registry ← sap[Registry]
    control: Control ← sap[Control]
    access: Access ← sap[Access]
IN  ...
```

Programm 19.3 Die Saps zu Abbildung 19.3

```
TYPE Query: Sap = {
  FUN request: Client × Certificate → Question → Beh Answer
}
TYPE Registry: Sap = {
  FUN hello: Client × Identification → Beh Challenge
  FUN certify: Client × Response → Beh Certificate
}
TYPE Control: Sap = {
  FUN authenticate: Client × Certificate → Beh Bool
}
TYPE Access: Sap = {
  FUN ask: Question → Beh Answer
}
```

Dabei hätten wir uns die doppelte Angabe der Typen jeweils sparen können. Aber in dieser Form wird deutlich, was der Konstruktor *sap* bewirkt: Er nimmt als Argument einen Typ der Klasse *Sap* und kreiert einen SAP dieses Typs.

Die *Clients*

Die einfachsten Agenten in dem Szenario aus Abbildung 19.3 sind die Clients; denn sie fragen nur Dienste an ohne selbst welche bereitzustellen. Programm 19.4 enthält die Definition der Clients.

Programm 19.4 Der Agententyp *Client*

```
STRUCTURE Client = {
  TYPE Client = Agent Void
  FUN client: Beh Void → Beh Client
  DEF client(op) = agent(op)                    -- kreiere neuen Agenten
  FUN register: Client × Identification → Registry → Beh Certificate
  DEF register(client, ident)(registry) =
      LET challenge ← registry.hello(client, ident)
          response = solve(challenge)
          certificate ← registry.certify(client, response)
      IN
          yield certificate
}
```

Das umfassende Programm enthält typischerweise folgende Fragmente:

$client_i \leftarrow client(\text{«}program_i\text{»})$

Im Programm «$program_i$» des Clients wird dann als Erstes die Registrierung durchgeführt; danach wird im Allgemeinen immer wieder der Service *request* am SAP *query* angefordert:

$$
\begin{array}{l}
\text{«}program_i\text{»} = \\
\quad \text{LET } client \leftarrow self \\
\qquad\quad certificate \leftarrow register(client, ident_i)(registry) \\
\quad \text{IN} \quad \ldots \\
\qquad\quad answer \leftarrow query.request(client, certificate)(question) \\
\qquad\quad \ldots
\end{array}
$$

Allerdings ist noch ein Problem zu lösen: Damit dies so programmierbar ist, müssen in «$program_i$» die SAPs *query* und *registry* bekannt sein! Das kann durch entsprechende Parametrisierung erfolgen. Der obige Konstruktor muss deshalb etwas adaptiert werden:

$\text{LET } client_i \leftarrow client(\text{«}program_i\text{»}(registry, query))$

und «$program_i$» ist eine Funktion der Form

$\text{«}program_i\text{»} = \lambda\, registry\colon Registry,\ query\colon Query \bullet \ldots$

Diese umständliche Notation wird uns dazu motivieren – analog zur Situation bei Objekten und Klassen – entsprechende syntaktische Erweiterungen einzuführen (s. Abschnitt 19.5).

Der *Guard*

Der Guard ist der einzige Agent seiner Art und fungiert sowohl als Client als auch als Server. Programm 19.5 enthält die entsprechende Definition; dabei wird die Operation *fail* aus der Struktur *TimeMonad* von Programm 17.3 benutzt.

Programm 19.5 Der Agent *Guard*

```
STRUCTURE Guard(query: Query, control: Control, access: Access) = {
  TYPE Guard = Agent Void
  FUN guard: Beh Guard
  DEF guard = agent( forever(query ⋉ request)) )
  FUN request: Client × Certificate → Question → Beh Answer
  DEF request(client, certificate)(question) =
      LET okay ← control.authenticate(client, certificate)
      IN
      IF okay THEN answer ← access.ask(question) & yield answer
              ELSE fail("not authorized")                    FI
}
```

Das Programm, das der Guard bei seiner Erzeugung mitbekommt, besagt, dass der Agent kontinuierlich am SAP *query* die Operation *request* anbietet. Die Implementierung dieser Operation fordert zuerst am SAP *control* die Operation *authenticate* an. Im Erfolgsfall wird dann am SAP *access* die Operation *ask* angefordert.

Das liest sich zwar einigermaßen elegant, versteckt aber ein hässliches technisches Problem: Die Funktion *request* bezieht sich auf die drei SAPs *query*, *control* und *request*, die als Parameter der Struktur übergeben werden. Damit wird ein fundamentales Problem „wegdefiniert“: Die drei SAPs werden erst zur Laufzeit (monadisch) generiert! Also kann die Struktur *Guard* erst *danach* erzeugt werden. Folglich muss es einen monadischen Konstruktor der folgenden Art geben:

```
LET query ← ...
    ...
    Guardian ← yield Guard(query, control, access)
    guard ← Guardian.guard
IN  ...
```

Dieses etwas umständliche Verfahren kennen wir schon von den Objekten aus Kapitel 18. Deshalb werden wir auch hier entsprechende notationelle Erleichterungen einführen (s. Abschnitt 19.5).

Der Agent *Authorization*

Der Agent *Authorization* bedient die beiden SAPs *registry* und *control*. Dies ist in der Struktur *Authorization* von Programm 19.6 gezeigt.

Die Registrierung erfolgt in zwei Schritten: Zuerst teilt der Client über den SAP *registry* mit dem Dienst *hello* seine Identification mit. Der Authorization-Server schickt daraufhin einen Challenge zurück (z. B. einen Textstring), den der Client lösen muss (z. B. mittels einer Hash-Funktion oder sonstigen Signaturtechnik). Diese Lösung wird über den SAP *registry* mit dem Dienst *certify* zurückgeschickt, worauf der Server ein Certificate ausstellt.

Dieses Programm enthält eine Reihe von Komplikationen, die wir im Folgenden der Reihe nach diskutieren wollen.

- Zunächst gilt ähnlich wie bei *Guard*, dass die Struktur erst zur Laufzeit generiert werden kann, wenn die SAPs *control* und *registry* vorliegen.
- Die *Map* bildet vom Typ *Client* in den Summentyp (*Challenge* | *Certificate*) ab. Dies erfordert einige Typtests und Typanpassungen, die wir hier um der besseren Lesbarkeit willen weggelassen haben.
- Das Programm des Agenten führt die Operation *run* aus. Aufgrund der Rekursion dieser Operation werden immer wieder die Dienste *hello*, *certify* und *authenticate* an den entsprechenden SAPs angeboten. Der Parameter *Map* enthält die aktuellen Informationen über alle angemeldeten Clients.

Programm 19.6 Der Agent *Authorization*

```
STRUCTURE Authorization(control: Control, registry: Registry) = {
  TYPE Authorization = Agent Void
  FUN authorization = Beh Authorization                    -- Konstruktor
  DEF authorization = agent(run(□))
  FUN run: Map → Beh Void                                  -- das "Programm"
  DEF run(map) =
        registry ⋉ hello(map) → newMap & run(newMap)
      + registry ⋉ certify(map) → newMap & run(newMap)
      + control ⋉ authenticate(map) & run(map)
  FUN hello: Map → Client × Identification → Beh(Challenge × Map)   -- Service
  DEF hello(map)(client, ident) =
      LET challenge = riddle(ident)
          newMap = map ⇐ (client ↦ challenge)
      IN
      yield (challenge, newMap)
  FUN certify: Map → Client × Response → Beh(Certificate × Map)   -- Service
  DEF certify(map)(client, response) =
      LET challenge = map(client)
          certificate = «suitable certificate»
          newMap = map ⇐ (client ↦ certificate)
      IN
      IF response fits challenge THEN yield (certificate, newMap)
                                 ELSE fail("invalid response")  FI
  FUN authenticate: Map → Client × Certificate → Beh Bool   -- Service
  DEF authenticate(map)(client, certificate) =
      yield (map(client) = certificate)
}
```

- Der wichtigste Aspekt dieses Programms ist aber, dass es die komplizierten Variationen der Services benötigt, bei denen zusätzliche Parameter und Resultate auftreten.
  - Beim Service *hello* wird das zusätzliche Argument *map* gebraucht, um den Challenge für den neuen Client vermerken zu können. Deshalb muss dem Server auch für die Continuation als zweiter Wert die geänderte Map zur Verfügung gestellt werden.
  - Beim Service *certify* gilt das Gleiche. Auch hier wird die Map sowohl als Argument als auch als zusätzliches Resultat gebraucht.
  - Bei *authenticate* wird die Map nur als Argument benötigt.

An diesem Programm sieht man auch den Effekt unserer etwas gewagten, aber für die Lesbarkeit sehr nützlichen Entwurfsentscheidung, für die Services komplexere Funktionalitäten zuzulassen. Um diesen Effekt noch einmal zu

sehen, betrachten wir den Service *hello*. (Man vergleiche dazu auch Tabelle 19.1.)

- Im SAP-Typ *Registry* ist der Dienst mit folgender Signatur charakterisiert:

  FUN $hello$: $Client \times Identification \to Beh\ Challenge$

- Im Client wird der Dienst dementsprechend in folgender Form nachgefragt:

  $\ldots\ registry.hello(client, ident) \to challenge\ \&\ \ldots$

- Im Server ist der Dienst folgendermaßen implementiert:

  FUN $hello$: $Map \to Client \times Identification \to Beh(Challenge \times Map)$
  DEF $hello(map)(client, ident) = \ldots$

- Im Server wird der Dienst in folgender Form am SAP angeboten:

  $\ldots\ registry \bowtie hello(map) \to newMap\ \&\ \ldots$

19.5 „Globale“ Agenten und SAPs

Bei Agenten und SAPs stoßen wir auf das gleiche notationelle Problem, das wir auch bei Objekten schon hatten. Der Zwang, sie monadisch kreieren zu müssen, führt manchmal zu unleserlichen Programmen; das wurde im Beispiel des vorigen Abschnitts 19.4 mehrmals deutlich. Dabei haben wir zwei verschiedene Vorgehensweisen gesehen:

- Man kann – wie bei der Struktur *Client* in Programm 19.4 – die benötigten SAPs jeweils den einzelnen Funktionen als Parameter mitgeben.
- Man kann aber auch – wie bei den Strukturen *Guard* und *Authorization* in den Programmen 19.5 und 19.6 – die gesamte Struktur mit den benötigten SAPs parametrisieren.

Beide Varianten haben Nachteile: Im ersten Fall hat man eine große Menge von Parametern, die einfach nur durchgereicht werden, was die Programme mit viel „*formal noise*“ aufbläht. Im zweiten Fall muss man mit dynamisch generierten Strukturen arbeiten, was die Lesbarkeit durch viele Selektionen belastet.

Wären die SAPs normale Werte, würden diese Probleme aufgrund der normalen Scoping-Regeln für Packages und Strukturen gar nicht entstehen. Deshalb erlauben wir die gleiche syntaktische Variation, die wir schon für Objekte benutzt haben. Da die entsprechenden Prinzipien der „globalen“ Objekte und Klassen schon im Abschnitt 18.1.2 erläutert wurden, beschränken wir uns bei SAPs und Agenten auf ihre Illustration durch Beispiele. Der SAP *registry*: *Registry* aus Programm 19.3 kann jetzt folgendermaßen definiert werden:

```
SAP registry = {
  FUN hello: Client × Identification → Beh Challenge
  FUN certify: Client × Response → Beh Certificate
}
```

Das subsumiert die entsprechende Typdeklaration sowie die Erzeugung mittels *registry* ← *sap*[*Registry*]. Diese Erzeugung erfolgt jetzt implizit beim Programmstart durch den Compiler.

Bei den Agenten wählen wir zur Illustration den *Guard* von Programm 19.5.

```
AGENT Guard = {
  EXEC forever(query ⋉ request)
  FUN request: Client × Certificate → Question → Beh Answer
  DEF request(client, certificate)(question) =
      LET okay ← control.authenticate(client, certificate)
      IN
      IF okay THEN answer ← access.ask(question) & yield answer
              ELSE fail("not authorized")                    FI
}
```

Wie man hier sieht, entfällt vor allen Dingen die komplexe Indirektion mit der Laufzeit-Generierung der Struktur *Guardian*, die durch die Parametrisierung mit den SAPs erzwungen wurde. Diese sind jetzt nach den üblichen Scoping-Regeln bekannt.

Allerdings kommt ein neues Feature hinzu. Da wir jetzt die Operation *guard* (die ja den Konstruktor *agent* realisiert) nicht mehr explizit hinschreiben, müssen wir dem Agenten sein Programm anderweitig mitgeben. Dazu dient das Schlüsselwort EXEC.

19.6 Spezialfälle: Kanäle und *Gates*

Die Service-Access-Points sind ein flexibles und mächtiges Werkzeug; aber in den verschiedenen praktischen Anwendungen gibt es unterschiedliche Bedürfnisse. Manchmal genügt etwas Einfacheres als SAPs, z. B. „Kanäle“, manchmal braucht man etwas Mächtigeres, z. B. „Gates“.

19.6.1 Kanäle

Kanäle sind eine besonders einfache Form von SAPs, bei der es nicht möglich ist, unterschiedliche Dienste anzufordern. Man kann nur auf der einen Seite Daten senden und diese auf der anderen Seite empfangen. Dies wird in der Literatur traditionell folgendermaßen notiert:

```
... c ! a ...          -- Senden des Wertes a auf Kanal c
... c? → x & ...       -- Empfangen eines Wertes von Kanal c
```

Wir sehen aber neben diesen Mixfix-Symbolen noch äquivalente Funktionsschreibweisen vor (die wir auch hier wieder mit dem „.“-Operator verbinden):

$\ldots c.put(a) \ldots$ −− *Senden des Wertes a auf Kanal c*
$\ldots c.get \rightarrow x \,\&\, \ldots$ −− *Empfangen eines Wertes von Kanal c*

Wie in Programm 19.7 gezeigt wird, lassen sich diese Operationen sehr leicht über SAPs implementieren. Ein Kanal ist ein SAP, der genau einen Dienst

Programm 19.7 Die Struktur der „Kanäle“

```
STRUCTURE Channel = {
  TYPE (Chan α): Sap                                  −− Typ
  FUN channel: Beh[Chan α]                            −− Konstruktor
  FUN get: Chan α → Beh α                             −− empfangen
  FUN _?: Chan α → Beh α                              −− empfangen (Postfixnotation)
  FUN put: Chan α → α → Beh α                         −− senden
  FUN _ ! _: Chan α × α → Beh α                       −− senden (Infixnotation)
  PRIVATE PART
    DEF Chan α = { FUN transmit: α → Beh α }          −− Typdefinition
    DEF channel = sap[Chan α]                         −− Konstruktor
    DEF get(c) = c ⋉ transmit                         −− empfangen
    DEF c? = get(c)                                   −−   (als Server)
    DEF put(c)(a) = c.transmit(a)                     −− senden
    DEF c ! a = put(c)(a)                             −−   (als Client)
    FUN transmit: α → Beh α                           −− Service-Implementierung
    DEF transmit(a) = yield a
}
```

anbietet, nämlich *transmit*. Allerdings ist diese Tatsache im *Private part* der Struktur verborgen. Von außen können Anwendungsprogramme für Kanäle nur die beiden Operationen *put* und *get* ausführen. Damit ist dort auch völlig unsichtbar, was ein Dienstangebot ist und was eine Dienstnachfrage; dadurch werden die Anwendungsprogramme einfacher zu schreiben und lesbarer. Trotzdem ist die Rollenverteilung hier interessant. Der Sender ist der Client, der den Dienst *transmit*(a) anfordert. Der Empfänger bietet als Server den Dienst *transmit* an (und erhält so die „Zusatzinformation“ a).

Was passiert, wenn mehrere Agenten auf einem Kanal chan senden und empfangen wollen? Das heißt, wir betrachten ein Szenario der folgenden Form:

Agent A: $\ldots chan\,!\,a \ldots$ *Agent X*: $\ldots chan\,?\ldots$
Agent B: $\ldots chan\,!\,b \ldots$ *Agent Y*: $\ldots chan\,?\ldots$

In dieser Situation ist weitgehend offen, wer welchen Wert erhält und in welcher Reihenfolge. Aufgrund der Fairness-Annahme von Abschnitt 19.3 hängt das ganz davon ab, wann die vier Agenten ihre jeweiligen Anfragen stellen.

19.6.2 *Gates*: SAPs + Agenten

Für viele Anwendungen – insbesondere für die graphischen Benutzerschnittstellen, die wir im folgenden Kapitel 20 betrachten werden – bieten die elementaren Agenten und SAPs eine viel zu niedere Abstraktionsebene. (Man könnte von einer „Assemblersprache der Kommunikation“ sprechen.) Es ist ein zentrales Konzept moderner Programmiersprachen, dass man in ihnen das Abstraktionsniveau auf höhere Stufen anheben kann – und für funktionale Sprachen ist das sogar eines der primären Anliegen.

Eine trotz ihrer Einfachheit sehr nützliche Abstraktion erhalten wir durch das Konzept der ***Gates***. Ein Gate ist im Wesentlichen die Kombination aus einem Agenten und mehreren SAPs (s. Abbildung 19.4).

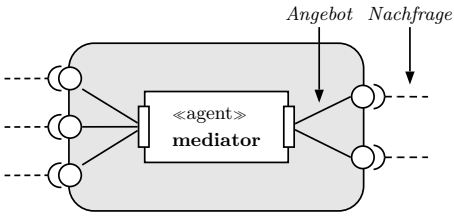


Abb. 19.4: Konzept eines Gates

Von außen betrachtet ist ein Gate also nichts anderes als eine Ansammlung von Service-Access-Points. Aber es gibt einige besondere Aspekte, die das Konzept sehr benutzerfreundlich machen:

- Die Agenten im Umfeld senden nur Dienst*nachfragen* an das Gate. Die (programmiertechnisch aufwendigeren) Dienst*angebote* erbringt der lokale Mediator.
- Mit Hilfe des lokalen Agenten sind komplexere Verarbeitungen möglich. Das heißt, in einem Gate kann sowohl eine *Vorverarbeitung* der Anfragen als auch eine *Nachbereitung* der Antworten stattfinden, sowie eine komplexere *Vermittlung* zwischen Clients und Servern.

Programmiertechnisch lassen sich Gates in zwei Formen programmieren, von denen die zweite softwaretechnisch gesehen die bessere ist.

- Man kann die einzelnen SAPs sichtbar lassen; dann kann allerdings auch jeder externe Agent die Dienste anbieten (nicht nur nachfragen).

- Man kann die Dienste der SAPs in eigene Operationen kapseln; dann kann kein anderer Agent als der interne Mediator die Dienste anbieten.

  Wir werden in den folgenden Beispielen beide Varianten präsentieren.

Beispiel: *Cache*

Als erstes Beispiel wollen wir einen *Cache* zwischen zwei SAPs platzieren wie in Abbildung 19.5 skizziert.

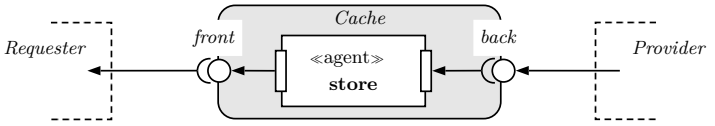


Abb. 19.5: Ein *Cache*

Programm 19.8 zeigt eine mögliche Implementierung dieses Designs.

Programm 19.8 Ein Cache

```
STRUCTURE Cache(α, β) = {
  TYPE Access: Sap = { FUN request: α → Beh β }
  FUN cache: Beh(Access × Access)
  DEF cache = LET front ← sap[Access]                    -- kreiere SAP
                  back ← sap[Access]                     -- kreiere SAP
                  store ← agent(run(□)(front, back))     -- kreiere Agent
              IN
                  yield(front, back)
  PRIVATE PART
    FUN run: Map(α, β) → Access × Access → Beh Void
    DEF run(map)(front, back) =   front ⋉ request(map, back) → newMap
                                & run(newMap)(front, back)
    DEF request: Map(α, β) × Access → α → Beh(β × Map(α, β))
    DEF request(map, back)(a) =
            IF a ∈ map THEN yield(map(a), map)
            IF a ∉ map THEN LET b ← back.request(a)
                                newMap ← map ⊕ (a ↦ b)
                            IN
                                yield (b, newMap) FI
  }
```

Der Typ beider SAPs ist *Access*; damit gibt es an beiden den Dienst *request*. Bei der Erzeugung eines Caches werden sowohl die beiden SAPs *front*

und *back* erzeugt (und als Ergebnis abgeliefert) als auch der Agent *store*, der allerdings nach außen verborgen bleibt. Dieser Agent bietet kontinuierlich am SAP *front* den Dienst *request* an.

Wenn am SAP *front* eine Anfrage *request*(a) gestellt wird und a schon in der Map enthalten ist, wird der zugehörige Wert b sofort zurückgegeben. Ansonsten wird die Anfrage an den zweiten SAP *back* weitergereicht. Das Paar ($a \mapsto b$) wird dann in der Map vermerkt und b wird zurückgegeben.

Man beachte die Subtilität in dieser Implementierung. Der Cache implementiert den Dienst *request* als Angebot für den SAP *front*. Aber er benutzt ihn auch am SAP *back*, wo er von einem anderen Agenten aus der Umgebung bereitgestellt werden muss.

Natürlich könnte man dieses Design auch variieren, z. B. indem man die beiden SAPs *front* und *back* nicht mit dem Cache zusammen generiert, sondern sie als Parameter an den Konstruktor *cache* übergibt. In diesem Szenario sind die SAPs außen schon unabhängig generiert worden und werden nur noch mit dem Cache verbunden.

Generell gilt für dieses Design jedoch, dass *keine* Abschirmung erreicht wird. Jeder Agent, der die SAPs kennt, kann dort ebenfalls die Dienste anbieten und nachfragen. Damit ist nicht sichergestellt (jedenfalls nicht ohne weitere Maßnahmen), dass das erwartete Cacheverhalten tatsächlich stattfindet.

Beispiel: *Buffer*

Unsere SAPs realisieren im Wesentlichen *synchrone* Kommunikation; das heißt, Client und Server koordinieren sich in einem Rendezvous, was zu entsprechendem Warten führen kann. Bei den Kanälen ist das nicht anders. Manchmal hätte man aber lieber *asynchrone* Kommunikation; das heißt, der Server schickt einen Wert und kann sofort weiterarbeiten. Und auch der Client kann ohne Verzögerung Werte abholen, sofern der Server welche bereitgestellt hat. In dieser Situation spricht man von einem *Puffer*.

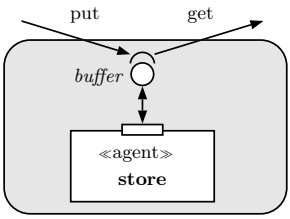


Abb. 19.6: Ein *Puffer*

Die Implementierung ist relativ einfach; sie ist in Programm 19.9 gezeigt. Das Gate stellt die beiden Operationen *get* und *put* bereit, die nichts anderes

Programm 19.9 Ein Puffer

```
STRUCTURE Buffer = {
  TYPE (Buffer α): Sap
  FUN buffer[α]: Beh(Buffer α)
  DEF buffer =      buf ← sap[Buffer α]                  -- kreiere den Buffer
                  & store ← agent(run(◊)(buf))           -- kreiere den Agenten
                  & yield buf                            -- Resultat ist der Buffer
  FUN get: Buffer → Beh α
  DEF get(buf) = buf.rcv → a & yield a
  FUN put: Buffer → α → Beh Void
  DEF put(buf)(a) = buf.snd(a) & done
  PRIVATE PART
    DEF Buffer = { FUN rcv: Beh α
                   FUN snd: α → Beh α }
    FUN run: Seq α → Buffer α → Beh Void
    DEF run(list)(buf) = buf ⋉ snd(list) → newList & run(newList)(buf)
                       +
                         buf ⋉ rcv | ( list ≠ ◊ ) → newList & run(newList)(buf)
    FUN snd: Seq α → α → Beh(α × Seq α)
    DEF snd(list)(a) = yield(a, list :· a)
    FUN rcv: Seq α → Beh α
    DEF rcv(list) = IF list ≠ ◊ THEN yield(ft list, rt list)
                               ELSE fail("buffer is empty") FI
}
```

tun, als die beiden Dienste *rcv* und *snd* am SAP *buffer* nachzufragen.

Der Konstruktor *buffer* kreiert sowohl einen SAP als auch einen Agenten *store*. Dieser Agent bietet kontinuierlich die Dienste *rcv* und *snd* am SAP an und verwaltet die Daten in einer Queue.

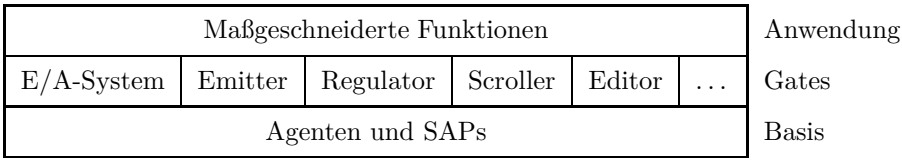
Da der interne Agent als Server für beide Dienste fungiert, können sowohl der schreibende als auch der lesende Agent als Client programmiert werden, was im Allgemeinen leichter und lesbarer ist.

Bei der Implementierung von *rcv* hätten wir uns den ELSE-Zweig sparen können, weil der Dienst ohnehin nur angefordert werden kann, wenn die Liste nicht leer ist. (Es ist also eine überflüssige – aber im Software-Engineering manchmal doch empfehlenswerte – Vorsichtsmaßnahme zur Steigerung der Robustheit des Codes.)

Durch dieses Design ist der SAP intern verborgen. Nachfragen können immer noch gestellt werden, indem die Operationen *get* und *put* aufgerufen werden. Aber kein anderer Agent kann die Dienste am SAP *buffer* anbieten. Wir haben also eine gute softwaretechnische Modularisierung und Abschirmung erreicht.

Weitere Beispiele: Spezielle Gates für GUIs

In graphischen Benutzerschnittstellen treten typischerweise eine Reihe von Gates auf, so dass wir folgende Architektur haben.



Dabei ist z. B. der Emitter nichts anderes als ein Kanal, auf dem als „Ticks“ einfache Signale kommen, während der Scroller in einem GUI-System die Koordination zwischen dem Fensterinhalt und dem Scrollbar an der Seite vornimmt. Ein Editor-Gate stellt sogar eine ganze Editoren-Funktionalität für ein Fenster bereit. In Kapitel 20 werden wir einige dieser Beispiele detaillierter studieren.

19.7 OPAL, CONCURRENT HASKELL, EDEN und ERLANG

Zum Abschluss wollen wir noch einen Blick auf die Integration von Agenten oder Prozessen in verschiedenen funktionalen Programmiersprachen werfen.

In OPAL ist das Konzept der SAPs effektiv implementiert, allerdings in einer älteren Variante. In dieser Variante ist ein SAP mit zwei Datentypen verbunden, einem für die Anfragen und einem für die Antworten. Dies wird (leicht an unsere Notationen adaptiert) in der folgenden Struktur *Service* beschrieben, die den Typ *Sap* der Service-Access-Points definiert und die Operationen zur Service-Nachfrage und zum Service-Angebot einführt.

```
STRUCTURE Service = {
  TYPE Sap[α, β]
  FUN sap: Beh(Sap(α, β))                    -- α=queries, β=answers
  DEF sap = «erzeuge neuen Service Access Point»
  FUN _ ? _: Sap(α, β) × α → Beh β
  DEF sap ? request = «Dienstnachfrage am SAP»
  FUN _ ! _: Sap(α, β) × (α → Beh β) → Beh β
  DEF sap ! offer = «Dienstangebot am SAP»
  FUN _ | _ ! _: Sap(α, β) × (α → Bool) × (α → Beh β) → Beh β
  DEF (sap | cond ! service) = «Angebot eines bedingten Dienstes»
}
```

Der Prozess erfolgt genauso Rendezvous-basiert wie wir ihn für unsere SAPs beschrieben haben. Aber jetzt werden nur Daten ausgetauscht; diese müssen in der Praxis fast immer als Elemente entsprechender Summentypen definiert werden.

Die bei uns direkt verwendeten Dienst-Operationen sind bei der OPAL-Variante dann über entsprechende Konstruktoren des Summentyps kodiert. Deshalb muss fast immer in der Continuation der SAP-Kommunikation eine Operation folgen, die mittels einer geeigneten musterbasierten Definition diese Dienste wieder einzeln betrachtet. Mit anderen Worten: Während bei uns die Dienste direkt programmiert werden, gibt es in der OPAL-Version jeweils große Funktionen, die die einzelnen Dienste per Fallunterscheidung extrahieren. Das macht in der Praxis die Programme komplexer und unleserlicher als bei uns.

CONCURRENT HASKELL verfolgt einen ähnlichen Ansatz wie unsere Agenten in Verbindung mit Kanälen. Auch hier werden Monaden zur Kapselung der Nebenläufigkeit verwendet. Die Agenten – hier *Prozesse* genannt – werden explizit durch eine Funktion *forkIO* gestartet. Die Interprozess-Kommunikation und Synchronisation der Prozesse erfolgt über so genannte *MVars*, gemeinsame veränderbare Sperrvariablen, die man als Channels ähnlich zu denen aus Kapitel 19.6.1 betrachten kann.

type MVar a
newMVar :: *IO* (*MVar a*)
takeMVar :: *MVar a* → *IO a*
putMVar :: *MVar a* → *a* → *IO* ()

Ein Wert vom Typ *MVar a* ist der Name einer solchen veränderbaren Variablen oder besser deren Lokation. Diese ist entweder leer oder sie enthält einen Wert vom Typ *a*. Mit Hilfe von drei primitiven Operationen kann man auf einer MVar arbeiten:

- *newMVar* erzeugt eine MVar.
- *takeMVar mvar* realisiert blockierendes Lesen. Ist die MVar *mvar* leer, blockiert die Operation, sonst wird ihr Wert ausgelesen und zurückgeliefert, und *mvar* wird leer hinterlassen.
- *putMVar mvar value* schreibt den Wert *value* in die MVar *mvar*. Gibt es durch *takeMVar* blockierte Prozesse, so wird einer von ihnen freigegeben und eine Kommunikation findet statt. Das Schreiben auf eine nichtleere MVar generiert einen Fehler, die Operation *putMVar* ist also nichtblockierend.

Bis auf den letzten Punkt verhalten sich die Funktionen *takeMVar* und *putMVar* ähnlich zu unseren Kanal-Operationen ? und !.[3] CONCURRENT HASKELL verzichtet außerdem ausdrücklich auf einen Choice-Operator; eine nichtdeterministische Auswahl kann hier aber mit Hilfe der MVars nachgebildet werden [116].

Im Gegensatz zu CONCURRENT HASKELL, das nebenläufige Prozesse auf einem Prozessor ausführt, unterstützt GLASGOW DISTRIBUTED HASKELL (GdH)

[3] Um das erneute Beschreiben nicht-leerer MVars zu verhindern, kann man auf eine zweite MVar zurückgreifen, die alternierend gelesen und beschrieben wird. Dabei nutzt man aus, dass *takeMVar* blockierend arbeitet.

[120] deren explizite Verteilung auf verschiedene Prozessorelemente. GdH ist eine Obermenge von CONCURRENT HASKELL und GLASGOW PARALLEL HASKELL (GpH) [94] und erlaubt daher auch die verteilte Berechnung mit Threads wie in GpH.

CONCURRENT HASKELL-Prozesse realisieren eine inhärent nebenläufige Struktur von Anwendungen, wie man sie in reaktiven Systemen vorfindet. In einem Texteditor möchte man z. B. neben Tastatureingaben auch Mouseclicks erkennen können. Prozesse zur nebenläufigen Steuerung solcher Teilaufgaben werden im Programm explizit erzeugt und kommunizieren miteinander durch den Austausch von Nachrichten. Im Unterschied dazu dienen GpH-Threads der Erhöhung der Geschwindigkeit bei der Programmauswertung. Im Programm kann durch *par*-Annotationen die parallele Auswertung von Teilaufgaben durch Threads vorgeschlagen werden, die parallele Maschine erzeugt solche Threads daraufhin in Abhängigkeit ihres aktuellen Zustands.

EDEN [25, 94] ist ebenfalls eine Erweiterung von HASKELL und stellt neben der funktionalen Sprache als reiner Berechnungssprache für sequenzielle Programme zusätzliche Konstrukte zur Spezifikation von Prozessen zur Verfügung; damit trennt sich EDEN von einer rein funktionalen Sichtweise.

In EDEN kann man *Prozessabstraktionen* ähnlich wie Funktionsabstraktionen definieren. Eine Prozessabstraktion bildet Eingabedaten der so genannten Inports auf Ausgabedaten der so genannten Outports ab. Durch Instanziierung einer Prozessabstraktion wird ein Prozess generiert, dabei bilden die In- und Outports Kommunkiatonskanäle zwischen Prozessen. Für jedes Ausgabedatum wird ein Thread generiert, der den entprechenden Ausdruck auswertet und das Ergebnis über den Outport versendet. Über die Kanäle können nur vollständig ausgewertete Datenobjekte kommuniziert werden, mit Ausnahme von Listen, die elementweise übertragen werden. Die Prozesse blockieren bis ihre Eingabedaten in dieser Form vorliegen.

Andere Sprachen, wie beispielsweise ERLANG [14], lassen einfach Seiteneffekte zu. In ERLANG wird Nebenläufigkeit durch Sprachkonstrukte zur Prozesserzeugung und zum Versenden und Empfangen von Nachrichten durch asynchronen Nachrichtenaustausch ermöglicht, wobei auch hier Prozesse selbst nebenläufig arbeiten können, während innerhalb eines Prozesses die Berechnungen sequenziell erfolgen. Bei der Generierung eines Prozesses kann man zusätzlich auch den Rechner angeben, auf dem der Prozess arbeiten soll. Auf diese Weise kann man in ERLANG Programme auch verteilt ablaufen lassen.

Weitere Sprachen, in denen man auf die eine oder andere Weise mit nebenläufigen oder verteilten Agenten arbeiten kann, findet man in [141, 66].

20

Graphische Schnittstellen (GUIs)

Das ist alles so schön bunt hier!
Nina Hagen (TV-Glotzer)

Jedes interessante Programmiersystem muss heute die Möglichkeit bereitstellen, *graphische Benutzerschnittstellen* – kurz ***GUIs*** – zu gestalten. Niemand ist mehr bereit, z. B. beim 8-Damen-Problem Ausgaben der Art

(4, 8, 1, 5, 7, 2, 6, 3)

zu akzeptieren, obwohl das (bei richtiger Interpretation) die gleiche Information enthält wie die Illustration in Abbildung 20.1:

Abb. 20.1: Eine Lösung des 8-Damen-Problems

In der Funktionalen Programmierung kommt noch der Ehrgeiz hinzu, auch diesen Bereich so elegant wie möglich zu gestalten – wobei allerdings die inhärent monadische Natur des Problems diesem Bestreben gewisse Grenzen setzt. Trotzdem ist es möglich, viel von den Designschwächen klassischer imperativer GUI-Beschreibungen (wie z. B. TCL/TK, JAVA oder .NET) zu überwinden.

Anmerkung: In imperativen und objektorientierten Sprachen ist es üblich, die Gestaltung des Layouts durch eine Abfolge von Anweisungen zu bewirken, die Fenster kreieren, in diesen Fenstern neue Unterfenster positionieren, Attribute setzen, zeichnen etc. Alle diese Aktivitäten lassen sich – wie bei imperativen Programmen üblich – mit beliebigen anderen Operationen mischen. Und manchmal ist diese Vermischung sogar unumgänglich. Damit werden die Programme sehr leicht unübersichtlich (und erinnern an den „Spagetticode" früherer, längst überwunden geglaubter Zeiten). Für die Ablaufsteuerung werden vor allem folgende Prinzipien eingesetzt:

- *Event dispatching (im ursprünglichen* XWINDOWS-*System).*
- *Objektorientiertes Message passing (z. B. in* SMALLTALK, NEXTSTEP*).*
- *Callbacks (z. B. in* MOTIF, TCL/TK*)*

Von diesen Verfahren entspricht die Idee der Callbacks am ehesten unserer Technik, die auf Agenten und Gates basiert.

Aufgrund der Vielfalt von technischen Details ist es unmöglich, hier eine umfassende Beschreibung eines GUI-Systems zu geben. (Allein die bloße Auflistung der GUI-Bibliotheksklassen von JAVA in [52] hat nahezu den doppelten Umfang unseres gesamten Buches.) Deshalb beschränken wir uns darauf, die grundlegende Konzeption anhand eines Beispiels zu vermitteln.

20.1 GUIs – ein Konzept mit drei Dimensionen

Bei der Definition von GUIs muss man drei Dimensionen miteinander in Beziehung setzen:

- Das *graphische Erscheinungsbild* der einzelnen GUI-Elemente.
- Die *geometrische und hierarchische Anordnung* der GUI-Elemente.
- Die *Interaktion* zwischen den GUI-Elementen und den Programmaktivitäten.

In Abbildung 20.2 wird dies für das einfache Beispiel eines (auf dem Bildschirm simulierten) Taschenrechners angedeutet.

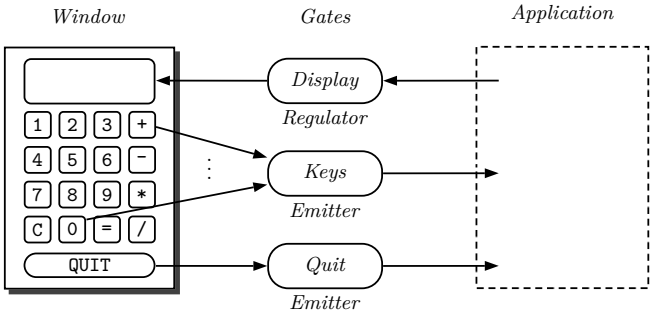


Abb. 20.2: GUI und Programm

Der Taschenrechner auf dem Bildschirm ist ein Bild, das aus graphischen Elementen zusammengesetzt ist (Displayfeld, Tasten und Quit-Button). Diese Elemente haben eine Gestalt (Farbe, Font etc.) Beim „Drücken“ der Tasten – also beim Anklicken mit der Mouse (oder auch über Tastatureingaben) – müssen entsprechende Effekte im Programm ausgelöst werden, die ihrerseits wiederum Änderungen im Bild hervorrufen.

Wir folgen hier im Wesentlichen dem Design des Systems OPALWIN [53, 54], wenn auch in modifizierter Form. (Auf andere Systeme gehen wir kurz am Ende des Kapitels ein.)

- Das „Bild“ ist (mindestens) ein Agent, der die Interaktion mit dem Benutzer übernimmt (mit Hilfe von Routinen des Betriebssystems).
- Das Anwendungsprogramm ist ein Agent bzw. eine Sammlung von Agenten.
- Die Interaktion zwischen diesen Agenten erfolgt über spezielle Gates.

In Programm 20.1 ist die Grundstruktur des Taschenrechners gezeigt. Sie folgt dem inzwischen klassischen *Model-View-Control*-Design; allerdings können aufgrund der extremen Einfachheit des Beispiels *Model* und *Control* miteinander verschmolzen werden; genauer: der *Control*-Anteil steckt implizit in der Interaktion über die Gates.

Programm 20.1 Die Struktur des Taschenrechner-Programms

```
PACKAGE Calculator = {
  AGENT Application = {...}     -- Model
  AGENT Window = {...}          -- View
  GATE Display = Regulator      -- Control
  GATE Keys = Emitter(Char)     -- Control
  GATE Quit = Emitter(Void)     -- Control
  }
```

Das Programm spiegelt genau das Design von Abbildung 20.2 wider: Es gibt eine Applikation (das *Model* inklusive *Control*), ein Fenster (den *View*) und die drei Gates *Display*, *Keys* und *Quit* zur Interaktion zwischen Fenster und Applikation.

20.2 Die Applikation (*Model*)

Das „Modell“ ist derjenige Teil des Systems, in dem die eigentlichen Berechnungen stattfinden. Der Zweck dieses separierten Designs ist es, die internen Berechnungen und Daten so unabhängig wie möglich von der Interaktion mit der Umwelt zu machen.

Programm 20.2 beschreibt das Modell, also die Ausführungslogik des Taschenrechners. Das Modell ist als globaler Agent definiert, der sofort beim

Programm 20.2 Das Modell

```
AGENT Model = {
  EXEC run(0, id)
  FUN run: Int × (Int → Int) → Action
  DEF run(acc, op) =
      key ← Keys.receive & continue(acc, op)(key)
      +
      quit ← Quit.receive & EXIT
  FUN continue: Int × (Int → Int) → Char → Action
  DEF continue(acc, op)(key)
      LET newAcc = IF digit(key)     THEN acc · 10 + asInt(key)
                   IF key = " = "    THEN op(acc)
                                     ELSE 0        FI
          newOp = IF digit(key) THEN op
                                ELSE operation(key)(acc) FI
      IN
      Display.set(asString(newAcc)) &          -- neuen Wert anzeigen
      run(newAcc, newOp)                       -- bereit für nächste Eingabe
  FUN operation: Char → Int → Int → Int
  DEF operation(" + ")(x) = ( x + _ )
  DEF operation(" − ")(x) = ( x − _ )
  DEF operation(" ∗ ")(x) = ( x · _ )
  DEF operation(" / ")(x) = ( x ÷ _ )
  DEF operation("c")(x) = ( id )
  DEF operation(" = ")(x) = ( id )
}
```

Programmstart generiert wird und seine Aktion *run* ausführt. Diese Aktion ist rekursiv so programmiert, dass sie läuft, bis das Quit-Signal eintrifft. (Wir verwenden das spezielle Schlüsselwort EXIT, um das Beenden des Programms zu charakterisieren.)

Die Operation *run* lauscht zunächst an zwei Emittern: Falls von *Keys* ein Wert kommt, wird die Arbeit mit der Operation *continue* fortgesetzt, falls von *Quit* ein Signal kommt, endet das Programm.

Der „Datenraum“ des Agenten umfasst nur zwei Elemente, nämlich den Akkumulator und die aktuell auszuführende Operation. Beide sind als Parameter der Funktion *run* (bzw. *continue*) realisiert.

Die eigentliche Ausführungslogik steckt in den beiden LET-Deklarationen.

- Wenn eine Ziffer eingegeben wird, dann wird sie an den Akkumulatorwert angefügt und die auszuführende Operation bleibt unverändert.

- Wenn ein Operatorsymbol wie z. B. `"+"` eingegeben wird, dann wird der Akkumulatorwert gelöscht; die neue Operation entsteht durch partielle Applikation der Addition auf den alten Akkumulatorwert.
- Bei der Eingabe von `"c"` (*clear*) wird der Akkumulator gelöscht und als Operation wird die Identität genommen.
- Bei der Eingabe von `"="` wird die aktuelle Operation auf den Akkumulatorwert angewandt, was den neuen Akkumulatorwert liefert. Die Operation wird auf die Identität gesetzt.

Zuletzt wird der neue Akkumulatorwert auf dem Display ausgegeben (genauer: an den entsprechenden Regulator geschickt) und die Operation *run* rekursiv aufgerufen.

Man sieht hier deutlich den gewünschten Effekt: Die Applikation kann die gesamte Kommunikation mit dem Benutzer ausschließlich über einige Dienst*nachfragen* erledigen. Das (programmiertechnisch komplexere) Angebot von Diensten ist nicht notwendig.

20.3 Graphische Gestaltung (*View*)

Die – vom Schreibaufwand her – aufwendigste der drei Dimensionen betrifft den Aspekt der *graphischen Gestaltung*. Jedes GUI-Element besitzt gewisse *Attribute*, die das GUI-Element konfigurieren. Aus der Fülle der heute üblichen Möglichkeiten seien nur einige der prominentesten Beispiele erwähnt:

- *die Farben* von Vordergrund und Hintergrund, wobei die Farbwahl in den beiden Zuständen „passiv“ und „aktiv“ verschieden sein kann;
- *die Fonts*, d. h. der Schriftsatz für Texte;
- *das Relief*, also ein 3-D-Effekt (*flat*, *sunken*, *raised*, *ridge*, *groove*);
- *der Text*, also die Beschriftung des GUI-Elements;
- *Bilder* (aus einer Datei) ebenfalls zur „Beschriftung“ von GUI-Elementen;
- *die Größe*, wobei man im Allgemeinen zwischen flexibler und fester Größe wählen und die Angaben in verschiedenen Maßeinheiten machen kann;
- *der Rand*, d. h. zusätzlicher Platz, der das GUI-Element innen bzw. außen jeweils horizontal oder vertikal umhüllt (*padX*, *padY*, *ipadX*, *ipadY*, ...);
- ...

Neben diesen Attributen, die das graphische Erscheinungsbild betreffen, gibt es noch Attribute, die im Zusammenhang mit dem *geometrischen Layout* gebraucht werden. Dazu gehören insbesondere

- *die Anordnung*: nebeneinander, übereinander, matrixartig etc.;
- *die Ausdehnung*: natürliche Größe, orientiert am umfassenden Element und den benachbarten Elementen etc.;
- *die Positionierung*: absolut, relativ zu anderen Elementen etc.;
- ...

Bevor wir diese Konzepte anhand einiger repräsentativer Beispiele diskutieren, wollen wir ihre Benutzung am Beispiel unseres Taschenrechners demonstrieren; sein graphisches Erscheinungsbild ist in Programm 20.3 angegeben (das größtenteils selbsterklärend ist).

Programm 20.3 Die graphische Gestaltung des Taschenrechners

```
AGENT Window = {
  EXEC window(view)                                                    -- Konstruktor
  FUN view: Compound
  DEF view = (display ⊟ keys ⊟ quit) with background(azure) +          -- übereinander
                                         foreground(black) +
                                         font(times(12)) +
                                         raised +
                                         ... +
                                         bindings
  FUN display: Label
  DEF display = label with background(white) +
                           foreground(blue) +
                           ...
                           bind(Display)
  FUN keys: View
  DEF keys = ⊞ ( (key*) * (("1", "2", "3",   " + "),                   -- matrixartig
                           ("4", "5", "6",   " − "),
                           ("7", "8", "9",   " * " ),
                           ("c", "0", " = ", " / " ) )
  FUN key: Char → Button
  DEF key(c) = button with text(c) +
                            background(blue) +
                            foreground(white) +
                            activeBackground(darkblue) +
                            ... +
                            bind(Keys)(c)
  FUN quit: Button
  DEF quit = button with background(blue) +
                          ... +
                          bind(Quit)
}
```

Der Konstruktor *window* bekommt als Argument ein GUI-Element, also die graphische Beschreibung des Fensters. In unserem Fall handelt es sich um ein zusammengesetztes Element (*Compound*), dessen geometrisches Layout durch folgenden Ausdruck festgelegt wird:

(*display* ⊟ *keys* ⊟ *quit*)

Der Operator ⊟ besagt, dass die entsprechenden Fensterelemente *übereinander* angeordnet werden. (Es gibt auch den analogen Operator ◫, der die Fenster *nebeneinander* anordnet.) Das so definierte zusammengesetzte Fenster[1] erhält durch die *with*-Klausel seine Attribute, also Farbe, Font, Rahmen, Größe etc. Diese Attribute werden an alle Unterfenster vererbt, sofern sie dort nicht redefiniert werden. Die durch *bindings* repräsentierten Attribute betrachten wir hier nicht weiter; sie betreffen z. B. die Aktionen zum Schließen oder Iconisieren des Fensters.

Das Fenster *display* ist vom Typ *Label* und wird durch den Konstruktor *label* definiert. Dabei legt die *with*-Klausel wieder die Attribute fest (durch die die globalen Attribute überschrieben werden). Der Ausdruck *bind*(*Display*) stellt die Verbindung zum Regulator-Gate *Display* her, über das das Fenster manipuliert werden kann.

Die Mächtigkeit des funktionalen GUI-Paradigmas zeigt sich gut am Beispiel der Funktion *keys*. Der Operator ⊞ definiert es als Fenster mit einem Matrix-Layout. Das Argument ist eine 4×4-Matrix von Buttons. Diese wird erzeugt, indem die Funktion *key* auf alle Elemente einer 4×4-Matrix von Zeichen angewandt wird. Die Funktion *bind*(*Keys*)(*c*) stellt dabei die Verbindung der Buttons zum (gemeinsamen) Emitter-Gate *Keys* her.

Damit wollen wir das Beispiel verlassen und uns den einzelnen Aspekten der GUI-Gestaltung zuwenden. Aber eines wird schon durch dieses winzige Beispiel überdeutlich: Für die graphische GUI-Beschreibung ist im Allgemeinen viel Programmcode hinzuschreiben, ohne dass irgendetwas intellektuell Spannendes passiert.

20.3.1 Arten von GUI-Elementen

Jedes GUI-System lebt davon, dass es dem Programmierer eine möglichst reichhaltige Auswahl an vordefinierten Fensterarten anbietet. Typischerweise gehören dazu etwa folgende Elemente:

- *Canvas*: eine „blanke“ Zeichenfläche;
- *Label*: ein nicht editierbares Textfeld;
- *TextField*: ein Feld zur Ein-/Ausgabe von einzeiligen Texten;
- *TextArea*: ein Feld zur Ein-/Ausgabe von mehrzeiligen Texten;
- *Button*: ein typischer „Druckknopf“;
- *Checkbutton*: ein Button, oft dargestellt als quadratische Box mit/ohne Haken oder als gedrückter/nichtgedrückter Knopf, womit die Zustände aktiviert/nicht aktiviert symbolisiert werden;
- *Radiobutton*: ebenfalls ein Button; im Unterschied zu Checkbuttons kann man in einer Liste von Radiobuttons genau einen aktivieren;
- *Scrollbar*: ein Balken zum Verschieben des aktuell sichtbaren Bereichs eines Textes oder Bildes;
- *Frame*: ein Rahmen;

[1] Solche zusammengesetzten Fenster werden z. B. in JAVA als `Container` bezeichnet.

- *Menü*: ein spezielles Window, das eine Auswahl von so genannten Menüelementen anbietet;
- *und viele andere mehr ...*

Diese reichhaltige Fülle von Features – die durch Attribute wie Font, Farbe etc. noch weiter anwächst – stellt die Implementierer vor eine zentrale Entwurfsentscheidung: *Soll der ganze Bereich stark oder schwach typisiert werden?* In den meisten Systemen hat man sich für eine schwache Typisierung entschieden, bei der vieles sogar erst zur Laufzeit geprüft wird. (Ein typisches Beispiel sind Farben, die oft nur als Strings repräsentiert werden.) JAVA und WINDOWS FORMS benutzen zumindest eine gewisse Vererbungshierarchie zwischen den Klassen für die GUI-Elemente.

Als Gegenpol zu diesen traditionellen Entwürfen werden wir hier zeigen, wie sich die Verwendung moderner Sprachmittel wie z. B. Typklassen auf das Design auswirken kann.[2]

Auf der obersten Ebene haben wir den Typ *Window*. Er steht für die eigenständigen Fenster, die auf dem Bildschirm angezeigt werden. Erzeugt werden diese Fenster durch die Operation

FUN *window*: α: *Form* $\rightarrow$ *Beh Window*

Wir kommen auf diese Operation und den Typ *Window* in Abschnitt 20.4.1 nochmals zurück. Jetzt interessieren wir uns für den Aufbau des Layouts.

Wir sammeln hier alle Arten von GUI-Elementen – also Buttons, Labels, Textfields usw. – in einer Typklasse *Form*, die in Programm 20.4 angegeben ist. (Die Operation *with* wird in den Abschnitten 20.3.2 und 20.3.3 erläutert.)

Programm 20.4 Die Typklasse der *Forms*

```
STRUCTURE Form = {
  TYPECLASS Form
  FUN _ with _ : α × Gestalt → α   VAR α: Form
  ...
}
STRUCTURE Button = { TYPE Button ... }
STRUCTURE Label = { TYPE Label ... }
...
```

Wir illustrieren die Definition von GUI-Elementen am Beispiel der Buttons. Programm 20.5 zeigt die Essenz der Realisierung. Der Typ *Button* ist ein Mitglied der Typklasse *Form*; er ist im Wesentlichen durch die Menge seiner Attribute charakterisiert. Diese Menge – die ja Elemente unterschiedlicher

[2] Aber trotzdem muss man sich über eines im Klaren sein: Die Art der Implementierung ist und bleibt eine Entwurfsentscheidung; man kann fast alles auch anders machen.

Programm 20.5 Die Struktur *Button*

```
STRUCTURE Button = {
  TYPE Button: Form = { text = Maybe Text,
                        font = Maybe Font,
                        background = Maybe Color,
                        foreground = Maybe Color,
                        activeBackground = Maybe Color,
                        activeForeGround = Maybe Color,
                        ...
                        padX = Maybe Real,
                        padY = Maybe Real,
                        bindings = Maybe(Set(Action))    }
  FUN button: Button                                    -- Konstruktor
  DEF button = { text = fail, ..., bindings = fail }
  FUN button: Text × Action → Button                    -- spezieller Konstruktor
  DEF button(txt, act) = button with text(txt) + bind(act)
}
```

Typen umfasst – wird als Gruppe mit entsprechenden Selektoren dargestellt. Die Attribute sind mittels *Maybe* optional gemacht; wenn ein Attribut nicht vorhanden ist (*fail*), dann wird ein entsprechender Defaultwert genommen. (Darauf kommen wir später noch einmal zurück.)

Man beachte, dass die einzelnen Arten von GUI-Elementen – Buttons, Labels, Textfields, Scrollbars etc. – sich in der Menge ihrer jeweiligen Attribute unterscheiden. Zwar sind eine Reihe von Attributen allen gemeinsam, z. B. *background*, *foreground* etc., aber einige sind doch spezifisch. Für dieses Problem gibt es zwei grundsätzliche Vorgehensweisen:

- Man kann die Obermenge aller vorkommenden Attribute bilden und diese für alle Typen der Klasse *Form* vorsehen. Die nicht passenden werden dann einfach ignoriert.
- Man kann für jeden Typ genau die sinnvollen Attribute vorsehen.

Aus Gründen der stärkeren Typprüfung wählen wir hier die zweite Variante, auch wenn sie schreibaufwendiger ist. (Aber dieser Aufwand ist ohnehin nur einmal beim Schreiben der Bibliothek zu leisten.) Bei dieser Variante könnte man die diversen Überlappungen noch durch ein ausgefeiltes System von Subklassen repräsentieren; das erscheint uns aber eher komplexitätssteigernd als hilfreich.

Die meisten dieser Attribute sind selbsterklärend. Die einzige Ausnahme ist das Attribut *bindings*. Es legt fest, wie der Button auf Ereignisse wie *keyPress*, *mousePress*, *mouseRelease*, *mouseClick* etc. reagiert. Darauf gehen wir in Abschnitt 20.4 noch im Detail ein.

Der Konstruktor *button* generiert einen Wert des Typs *Button*; da keine Attribute vorliegen, werden ihre Defaultwerte genommen (s. Abschnitt 20.3.4).

Für den einfachsten Spezialfall gibt es einen Konstruktor $button(txt, act)$, der einen Text für die Beschriftung und eine Aktion für das Klicken vorsieht.

Die anderen Arten von GUI-Elementen werden ganz analog beschrieben, weshalb wir hier auf ihre Angabe verzichten können.

20.3.2 Stil-Information

Im Beispiel unseres Taschenrechners haben wir gesehen, dass die einzelnen GUI-Elemente mit Hilfe der *with*-Klausel attributiert werden können:

$$\begin{aligned}\text{DEF } key(c) = button\ with\ & text(c) + \\ & background(blue) + \\ & foreground(white) + \\ & activeBackground(darkblue) + \\ & \ldots\ + \\ & bind(Keys)(c)\end{aligned}$$

Dazu verwenden wir die Funktion *with*, die in Programm 20.4 folgendermaßen charakterisiert wurde:

$$\text{FUN } \_\,with\,\_\colon\ \alpha \times Gestalt \to \alpha \quad \text{VAR } \alpha\colon \mathit{Form}$$

Der Typ *Gestalt* dient also dazu, Attribute aufzusammeln, um diese dann in das GUI-Element eintragen zu können. Programm 20.6 zeigt die entsprechende Struktur. (Man kann dies effizienter implementieren, aber wir wollen hier vor allem eine konzeptuell klare Beschreibung.)

Der Typ *Gestalt* repräsentiert in der Tat die Obermenge aller im GUI-System vorkommenden Attribute. Diese werden wieder in Form einer großen Gruppe mit entsprechenden Selektoren dargestellt, wobei jedes Element mittels *Maybe* optional gemacht wird.

Zu jedem Attribut gibt es eine entsprechende Funktion, die dieses Attribut setzt. Das heißt z. B. bei $foreground(green)$, dass die Komponente *foreground* den Wert *green* hat, während alle anderen Komponenten den Wert *fail* haben. Bei der Operation $s_1 + s_2$ werden die beiden Gruppen so verschmolzen, dass jeweils der definierte Wert über *fail* dominiert. Wenn beide definiert sind, wird der von s_2 genommen. (Die Operation ist also *nicht kommutativ.*)

Bei der Operation (*form with style*) aus Programm 20.4 werden alle Attribute des Elements *form* mit den entsprechenden Werten aus *style* überschrieben, sofern diese nicht *fail* sind. Damit ist auch geklärt, dass in einer Anwendung der Art

$$\ldots((form\ with\ style_1)\ with\ style_2)\ldots$$

die Attributwerte aus $style_2$ diejenigen von $style_1$ überschreiben.

20.3.3 Geometrische Anordnung

Ein Fenster enthält im Allgemeinen viele GUI-Elemente, die in geeigneter Weise relativ zueinander positioniert werden müssen. In den gängigen Syste-

Programm 20.6 Die Struktur *Gestalt*

```
STRUCTURE Gestalt = {
  TYPE Gestalt = { text = Maybe Text,
                   font = Maybe Font,
                   background = Maybe Color,
                   activeBackground = Maybe Color,
                   ...
                   width = Maybe Real,
                   height = Maybe Real,
                   padX = Maybe Real,
                   padY = Maybe Real,
                   bindings = Maybe(Set(Action))    }
  FUN ∅: Gestalt
  FUN _ + _: Gestalt × Gestalt → Gestalt
  FUN text: Text → Gestalt
  FUN font: Font → Gestalt
  FUN background: Color → Gestalt
  ...
}
```

men besteht dieses Layout letztlich aus einer „Boxen-Welt“ (wie sie auch dem Textsystem TEX von Don Knuth zugrunde liegt).[3] Deswegen verwenden wir als wesentliche Funktionen die Komposition von Boxen, die in Programm 20.7 beschrieben ist.

Programm 20.7 Die Struktur *Layout*

```
STRUCTURE Layout = {
  TYPE Compound: Form
  FUN _ ◫ _: α × β → Compound   VAR α: Form, β: Form   -- horizontal
  FUN _ ⊟ _: α × β → Compound   VAR α: Form, β: Form   -- vertikal
  FUN _ ⊞ _: Matrix α → Compound   VAR α: Form        -- matrixartig
    ...
  FUN expand: Weight → α → α   VAR α: Form            -- strecken
}
```

[3] Hier unterscheidet sich unser funktionaler Ansatz deutlich von dem gängiger Systeme wie JAVA AWT/SWING, WINDOWS FORMS, TCL/TK und ähnlichen. Bei diesen muss man im Allgemeinen zuerst einen „Container“ kreieren, in den dann die weiteren Komponenten eingefügt werden. Dabei überraschen die teilweise recht undurchschaubaren Regeln der automatischen Größen- und Positionsberechnung den Programmierer immer wieder mit verblüffenden Effekten.

Mit diesen Operatoren kann z. B. das aus JAVA bekannte `BorderLayout` (vgl. Abbildung 20.3(a)) durch folgenden Ausdruck beschrieben werden:

$$\ldots\, north \boxminus (west \boxplus center \boxplus east) \boxminus south \ldots$$

Dabei ergibt sich aber ein Problem mit den Größen. In Abbildung 20.3 ist dieses Problem illustriert.

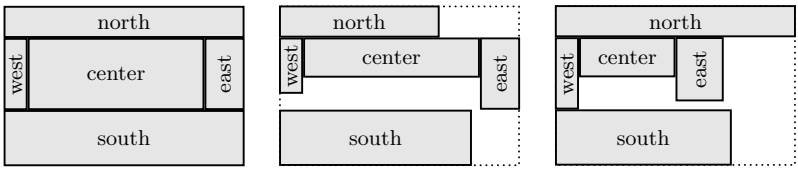


Abb. 20.3: Anordnung und Größe graphischer Elemente

Teilabbildung (a) entspricht dem, was man gerne sehen würde. Die Teilabbildungen (b) und (c) zeigen zwei verschiedene Szenarien, wenn die „natürlichen“ Größen der fünf Komponenten nicht zusammenpassen. Um in diesen Fällen trotzdem das Erscheinungsbild von (a) zu erhalten, brauchen wir die Operation *expand*. Mit den Notationen

$$\begin{array}{ll} \overrightarrow{e} = expand(1,0)(e) & -- \textit{ horizontal strecken} \\ \downarrow e = expand(0,1)(e) & -- \textit{ vertikal strecken} \\ \dot{\uparrow} e = expand(1,1)(e) & -- \textit{ beide strecken} \end{array}$$

lässt sich dann das Aussehen von Teilabbildung (a) in jedem Fall erzwingen:

$$\ldots \overrightarrow{north} \boxminus (\downarrow west \boxplus \dot{\uparrow} center \boxplus \downarrow east) \boxminus \overrightarrow{south} \ldots$$

Die Gewichte bei der *expand*-Operation bestimmen, wie der überschüssige horizontale und vertikale Platz auf die einzelnen GUI-Elemente aufgeteilt wird. Man beachte übrigens, dass das zusammengesetzte GUI-Element selbst durch eine Klausel der Art

$$\ldots (\overrightarrow{north} \boxminus (\downarrow west \boxplus \dot{\uparrow} center \boxplus \downarrow east) \boxminus \overrightarrow{south}) \ \textit{with} \ \begin{array}{l} width(\ldots) + \\ height(\ldots) + \\ \ldots \end{array}$$

eine Größe erhalten kann, an die dann die inneren Elemente angepasst werden müssen.

Aus pragmatischen Gründen gibt es für die Gestaltung zusammengesetzter GUI-Elemente noch einige Aspekte, die man berücksichtigen sollte:

- Man benötigt eine einfache Programmierung für häufige Spezialfälle, z. B. array- oder matrixförmige Anordnungen mit gleich großen Komponenten oder zumindest mit größenmäßig passenden Komponenten (wie in Abbildung 20.3 illustriert).

- Man muss Alignments wie *left*, *right*, *top*, *bottom* oder *centered* angeben können.
- Man muss – im anderen Extrem – auch in der Lage sein, Komponenten nahezu beliebig positionieren zu können.
- Man sollte auch in der Lage sein, in jeder beliebigen Komponente (nicht nur in den dafür prädestinierten *Canvas*-Elementen) frei zeichnen zu können.
- ...

Hier ist nicht der Platz, um ein solches Szenario in allen Facetten präsentieren zu können. Genau genommen handelt es sich sogar um ein offenes Forschungsthema. Um einen guten Entwurf machen zu können, müsste man als Erstes bestehende Standardsysteme, insbesondere JAVA AWT/SWING, WINDOWS FORMS, TCL/TK etc., analysieren und dabei die historischen Zufälligkeiten mancher Features durch ein systematisch bereinigtes Design ersetzen. Darüber hinaus sollte man sich auch die Erfahrungen aus weit entwickelten Text- und Layoutsystemen wie z. B. TEX/LATEX oder POSTSCRIPT zunutze machen, die viele Fragen wesentlich systematischer angegangen sind, als dies in den GUI-Systemen gemacht wurde.

Aus einer solchen Analyse kann man ein geschlossenes, homogenes und praktikables Design entwickeln, das dann „nur“ noch in funktionale Syntax gegossen werden muss. (Dass dies zu einem eleganteren Design führt, als man es aus den klassischen GUI-Systemen kennt, sollte an den obigen Beispielen deutlich geworden sein.)

20.3.4 *The Big Picture*

Wie sollte eine GUI-Bibliothek beschaffen sein? Eine offensichtliche Antwort ist, dass sie aus Packages und Subpackages bestehen sollte, die man gezielt importieren kann. Wie man so etwas thematisch organisiert, lässt sich ganz gut z. B. in JAVA SWING sehen (wenn man mit einigen historischen Zufälligkeiten aufräumt).

Eine weitere wichtige Frage betrifft die *Defaultwerte*. Dieser Aspekt tritt gerade in funktionalen Programmen deutlich zutage. (In objektorientierten Programmen kann man sich mit geeigneten Tricks, vor allem durch Redefinitionen, aus der Affäre ziehen – was allerdings nicht zur Qualität der Programme beiträgt.)

In jedem Fenstersystem hat man ein gewisses Standarddesign, z. B. Vorder- und Hintergrundfarbe, Font, Rahmen etc., das durch geeignete Defaultwerte durchgängig eingehalten werden sollte. Die Frage ist, wo und wie man diese Defaultwerte vorgeben kann.

Die eleganteste und den funktionalen Prinzipien angemessenste Lösung besteht sicher darin, die Packages und Strukturen der Bibliothek in *parametrisierter Form* zu definieren. Damit wäre es dann in Anwendungsprogrammen möglich, folgenden Aufbau zu realisieren:

```
STRUCTURE MyGeneralStyle = { ... «Defaultwerte» ... }
STRUCTURE MyButtonStyle = { ... «Defaultwerte» ... }
IMPORT Gui.Elements(MyGeneralStyle) WITHOUT Button
IMPORT Gui.Elements.Button(MyButtonStyle)
...
```

Natürlich sollte auch die Bibliothek selbst einen Default vorsehen. Der kann dann entsprechend in unparametrisierten Packages verfügbar gemacht werden (die – mittels Overloading – sogar genauso heißen können wie ihre parametrisierten Gegenstücke):

```
PACKAGE Gui.Elements = Gui.Elements(DefaultStyle)
```

Dabei definiert *DefaultStyle* die in der Bibliothek vordefinierten Defaultwerte.

20.4 Interaktion mit der Applikation (*Control*)

Die Interaktion zwischen dem GUI und der Anwendungslogik realisieren wir – wie in Abbildung 20.2 für das Beispiel des Taschenrechners illustriert – durch Gates (s. Abschnitt 19.6). Dementsprechend müssen sowohl das Fenster als auch die Applikation als Agenten realisiert werden.

Es gibt eine Reihe von Standard-Gates, die ausreichen, um für fast alle Arten von GUI-Elementen die Interaktion mit dem Programm zu bewerkstelligen. Dazu zählen vor allem *Emitter*, *Regulator* und *Selektor*, auf die wir gleich in Abschnitt 20.4.2 bis 20.4.4 genauer eingehen werden. Zuvor betrachten wir aber noch kurz das Fenster selbst.

20.4.1 Das Fenster als Agent

Jedes Top-level-Fenster auf dem Bildschirm korrespondiert zu einem entsprechenden Agenten im Programm. Dieser Agent wird durch die Operation *window* aus Programm 20.8 erzeugt.

Programm 20.8 Die Struktur *Window*

```
STRUCTURE Window = {
  TYPE Window = Agent Void
  FUN window: α → Beh Window   VAR α: Form                    -- Konstruktor
  DEF window(form) = «kreiere Agenten mit graphischer Erscheinung form»
}
```

Mit der Operation *window*(*form*) wird ein sehr spezieller Agent kreiert. Bei seiner Erzeugung zeigt er sofort das GUI-Element *form* auf dem Bildschirm an

(sofern dieses nicht gerade *invisible* oder *iconized* gesetzt ist). Danach durchläuft er eine Schleife, die typisch ist für alle modernen GUI-Treiber: Er lauscht permanent auf zwei Arten von Ereignissen:

- *Benutzeraktionen* wie z. B. Mousebewegung oder Mouseclick, Drücken einer Taste oder Bewegungen an einem Joystick;
- *Anfragen an den Gates*, die aus dem Anwendungsprogramm kommen.

Diese Ereignisse werden den zugehörigen GUI-Elementen zugeordnet. So wird z. B. bei einem Mouseclick anhand der Position des Cursors auf dem Bildschirm errechnet, welches GUI-Element gemeint ist; bei Tastatureingaben ist dasjenige Element betroffen, das gerade den „Fokus“ hat; usw.

Solche Ereignisse führen meistens dazu, dass der Window-Agent über die Gates entsprechende Nachrichten an das Anwendungsprogramm schickt, das seinerseits oft wieder mit Anforderungen an den Window-Agenten reagiert. Dies wird detaillierter in den folgenden Abschnitten dargestellt.

20.4.2 Emitter als Kanäle

Ein ***Emitter*** (vgl. Abbildung 20.4) überträgt Daten zwischen Agenten. In unserer Anwendung sind das beispielsweise Button-Klicks oder auch Tastaturanschläge, die so vom Window an die Anwendungslogik übergeben werden.

Wir betrachten wieder unser Beispiel des Taschenrechners aus dem Programm 20.1. Dort werden zwei Emitter eingeführt:

GATE $Keys = Emitter(Char)$

GATE $Quit = Emitter(Void)$

Im Applikationsprogramm 20.2 werden die beiden Emitter verwendet, um die entsprechenden Aktionen abzufragen:

$key \leftarrow Keys.receive \,\&\, \ldots$
$+$
$quit \leftarrow Quit.receive \,\&\, \ldots$

Schließlich wird in Programm 20.3 mittels $bind(\ldots)$ die Verbindung zwischen den Emittern und den zugehörigen GUI-Elementen hergestellt:

DEF $key(c) = button\ with\ \ldots + bind(Keys)(c)$

DEF $quit = button\ with\ \ldots + bind(Quit)$

Das Zusammenspiel dieser drei Aspekte soll im Folgenden erläutert werden. Generell haben wir eine Situation wie Abbildung 20.4 skizziert.

Warum braucht man hier ein Gate mit einem weiteren Agenten? Hätte es nicht auch ein reiner SAP getan? Dafür gibt es mehrere Gründe:

- Zunächst zur Erinnerung: Das gesamte Fenster ist selbst ein Agent. Würde dieser Agent z. B. beim Klicken auf den Button nur auf einem Kanal *chan* senden, dann wäre er blockiert, bis die Applikation den Kanal ausliest.

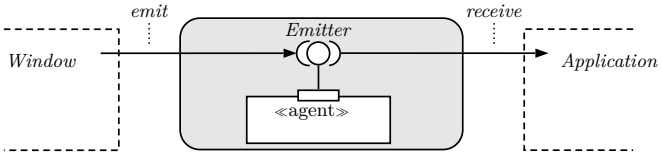


Abb. 20.4: Ein *Emitter-Gate*

Aber der Agent „Fenster“ muss stets bereit sein, auf Benutzeraktivitäten zu reagieren. Also delegiert er die Aufgabe des Sendens an die Applikation eines dazu geschaffenen parallel laufenden Hilfsagenten.

- Das Applikationsprogramm sollte möglichst wenig durch die technischen Kommunikationsdetails belastet werden. Und Gates sind – wie bereits erwähnt – notationell eleganter als reine SAPs, weil man sich bei ihnen auf Anfragen beschränken kann.
- Ein Gate bewirkt auch eine bessere softwaretechnische Kapselung der erlaubten Interaktionen. (Allerdings ist das keine hundertprozentige Abschirmung. Es ist z. B. immer noch möglich, aus der Applikation heraus ein *emit* auszulösen und damit einen Mouseclick vorzutäuschen – was manchmal durchaus gewollt sein kann.)

Programm 20.9 enthält die Definition der Struktur *Emitter*. Emitter sind

Programm 20.9 Die Struktur *Emitter*

```
STRUCTURE Emitter = {
  TYPE Emitter α                          -- Typ
  FUN emitter[α]: Beh(Emitter α)          -- Konstruktor
  FUN emit: Emitter α → α → Beh Void      -- senden
  FUN receive: Emitter α → Beh α          -- empfangen
  PRIVATE PART
    DEF Emitter = Chan                    -- Typdefinition
    DEF emitter = channel                 -- Konstruktordefinition
    DEF emit(e)(x) = agent(e ! x)         -- senden
    DEF receive(e) = e ?                  -- empfangen
}
```

im Wesentlichen asynchrone Kanäle. Diese Tatsache ist allerdings in der Implementierung verborgen, so dass aus der Umgebung heraus nur die beiden sichtbaren Operationen *emit* und *receive* benutzt werden können. Denn es kommt ein wichtiger Aspekt hinzu: Bei *emit* wird das Senden auf dem Kanal an einen neu geschaffenen Agenten delegiert, so dass der aufrufende Agent sofort weiterarbeiten kann. Der neue Agent verschwindet wieder, sobald das

Senden stattgefunden hat.[4] Aufgrund der Fairnessannahme für SAPs bleibt die Reihenfolge der Mouseclicks aus Sicht der Anwendung erhalten.

Ein Emitter kann an gewisse graphische Elemente wie z. B. Buttons als Attribut gebunden werden. Jedesmal wenn der Button „betätigt“ wird – also ein Mouseclick stattfindet während der Cursor über dem Button ist –, wird der Emitter getriggert. Als Verallgemeinerung kann ein Emitter auch an ein Event (s. Kapitel 20.4.5) eines Graphik-Elements gebunden werden.

FUN *bind*: *Emitter Void* → *Gestalt*
FUN *bind*: *Emitter* $\alpha \times \alpha \rightarrow$ *Gestalt*
FUN *bind*: *Event* × *Emitter* $\alpha \times \alpha \rightarrow$ *Gestalt*

Wenn wir also in unserem Taschenrechner einen Button mit folgendem Ausdruck kreiert haben (für einen gegebenen Emitter *Keys*):

$\ldots$ *button with* $\ldots$ + *bind*(*Keys*)(*c*)

dann führt der Fensteragent jedesmal, wenn dieser Button angeklickt wird, die folgende Operation aus:

emit(*Keys*)(*c*)

Das führt – gemäß der Definition in Programm 20.9 – dazu, dass ein anonymer Agent kreiert wird, der den Wert *c* an die Applikation sendet.

20.4.3 Regulator-Gates

Im Gegensatz zu Emittern, die eigentlich nur Kanäle sind, stellen Regulatoren komplexere Formen von Gates dar. Ein ***Regulator-Gate*** repräsentiert im Wesentlichen die *Gestalt* des zugehörigen GUI-Elements.

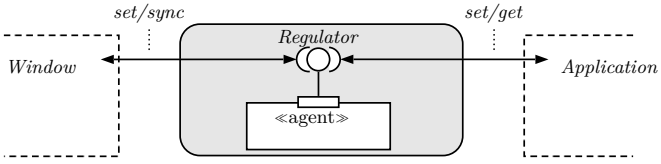


Abb. 20.5: Ein *Regulator-Gate*

Über ein Regulator-Gate (vgl. Abbildung 20.5) erhält die Applikation die Möglichkeit, die Attribute des GUI-Elements abzufragen und zu setzen, und das Fenster erhält die Möglichkeit, die (vom Benutzer ausgelöste) Änderung von Attributen mitzuteilen und seinerseits das Aussehen auf dem Bildschirm an die (aus der Applikation heraus) geänderten Attribute anzugleichen.

[4] Dieses Konzept verlangt, dass das Erzeugen und das Vernichten von Agenten „billig“ sein muss. Dies ist eine (heute durchaus erfüllbare) Herausforderung an den Compiler.

Auch hier orientieren wir uns wieder an unserem Taschenrechner-Beispiel. In Programm 20.1 wird ein Regulator-Gate eingeführt:

GATE *Display* = *Regulator*

Im Applikationsprogramm 20.2 wird der Regulator verwendet, um den Text für das Display zu setzen:

... *Display*.*set*(*asString*(*newAcc*)) ...

Schließlich wird in Programm 20.3 mittels *bind*(...) die Verbindung zwischen dem Regulator und dem GUI-Elementen hergestellt:

DEF *display* = *label with* ... + *bind*(*Display*)

Programm 20.10 definiert die Struktur *Regulator*. Angesichts der Komplexität der Aufgabe – schließlich geht es hier darum, die Bildschirmdarstellung im Gleichklang mit der Applikation zu halten – ist es nicht überraschend, dass diese Struktur etwas komplizierter ist.

Von außen ist ein Regulator mit den drei Operationen *get*, *set* und *sync* ansprechbar. Mit den ersten beiden können Agenten in der Umgebung die Konfiguration auslesen oder neu setzen; die dritte Operation *sync* kann zur Aktualisierung der Bildschirmdarstellung benutzt werden. (In JAVA wird dies als `repaint` bezeichnet.) Diese drei Operationen dienen (wie bei Gates üblich) dazu, die entsprechenden Dienste an dem zugehörigen internen SAP zu verbergen.

Der Regulator selbst besteht aus einem SAP und einem anonymen Agenten. Beide werden gemeinsam durch die Konstruktoroperation *regulator* kreiert, aber nur der SAP ist als Ergebnis sichtbar. Die einzelnen Operationen werden folgendermaßen realisiert:

- *get*: Am internen SAP wird der Dienst *query* nachgefragt. (Aus formalen Gründen wird dazu ein Argument benötigt; dafür bietet sich *void* an.) Dieser Dienst wird durch den internen Agenten kontinuierlich angeboten, der beim Rendezvous die aktuelle Gestalt übergibt.
- *set*: Am internen SAP wird der Dienst *change*(*newGestalt*) nachgefragt. Auch dieser Dienst wird kontinuierlich vom internen Agenten angeboten, der beim Rendezvous das Argument *newGestalt* übernimmt und zu seiner aktuellen Version macht; dabei wird die Versionsnummer erhöht.
- *sync*: Am internen SAP wird der Dienst *adjust* nachgefragt, der ebenfalls kontinuierlich vom internen Agenten am internen SAP angeboten wird. Das Rendezvous findet allerdings nur statt, wenn der Regulator eine neuere (= größere) Version besitzt. In diesem Fall werden sowohl die aktuelle Gestalt als auch ihre Versionsnummer zurückgeliefert.

Um die Funktionsweise eines Regulator-Gates vollständig zu verstehen, müssen wir seine Bindung an ein zugehöriges GUI-Element betrachten. In unserem Taschenrechner hatten wir folgende Situation:

DEF *display* = *label with* ... + *bind*(*Display*) + ...

Programm 20.10 Die Struktur *Regulator* (vereinfacht)

```
STRUCTURE Regulator = {
  TYPE Regulator: Sap                                          -- Typ
  TYPE Version = Int                                           -- Hilfstyp
  FUN regulator: Beh Regulator                                 -- Konstruktor
  FUN get: Regulator → Beh Gestalt                             -- Gestalt holen
  FUN set: Regulator → Gestalt → Beh Void                      -- neue Gestalt setzen
  FUN sync: Regulator → Version → Beh(Gestalt × Version)       -- Bild zeigen
  PRIVATE PART
    DEF Regulator = { FUN query: Void → Beh Gestalt            -- Typdefinition
                      FUN change: Gestalt → Beh Void
                      FUN adjust: Version → Beh(Gestalt × Version) }
    DEF regulator = LET reg ← sap[Regulator]                   -- Konstruktor
                        controller ← agent(run(reg, ∅, 1))
                    IN  yield reg
    DEF get(reg) = reg.query(void)
    DEF set(reg)(newGestalt) = reg.change(newGestalt)
    DEF sync(reg)(version) = reg.adjust(version)
    FUN run: Regulator × Gestalt × Version → Beh Void
    DEF run(reg, gestalt, version) =
          reg ⋉ query(gestalt)                  & run(reg, gestalt, version)
          +
          reg ⋉ change → newGestalt             & run(reg, newGestalt, version + 1)
          +
          reg ⋉ adjust(gestalt) | (< version)   & run(reg, gestalt, version)
    FUN query: Gestalt → Void → Beh Gestalt
    DEF query(gestalt)(void) = yield(gestalt)
    FUN change: Gestalt → Beh(Void × Gestalt)
    DEF change(newGestalt) = yield(void, newGestalt)
    FUN adjust: Gestalt → Version → Beh((Gestalt × Version) × Void)
    DEF adjust(gestalt)(version) = yield((gestalt, version), void)
}
```

Wenn der Fenster-Agent das GUI-Element *display* generiert, erzeugt er zugleich einen Agenten, der von da an eine Kontrollschleife ausführt (bis das GUI-Element verschwindet):

$$\ldots\ agent(loop(Display,\ display,\ 1)\ \ldots$$

Diese Kontrollschleife sieht für alle Arten von GUI-Elementen gleich aus.

```
FUN loop: Regulator × α: Form × Version → Beh Void
DEF loop(reg, form, version) =
      (newGestalt, newVersion) ← sync(reg)(version) -- abgleichen
    & «repaint(form, newGestalt)»                    -- zeichnen
    & loop(reg, form, newVersion)                    -- wieder bereit
```

Im Endeffekt ist dieser Agent bei der Operation *sync* blockiert, bis die Versionsnummer am Regulator größer geworden ist als die eigene. Das bedeutet, dass irgendeine Änderung stattgefunden hat. In diesem Augenblick wird die Blockierung aufgehoben, das *repaint* findet statt und der Agent wird wieder blockiert.

20.4.4 Weitere Gates

Neben den oben diskutierten Standardgates gibt es noch weitere, teilweise sehr spezialisierte Gates, die jeweils die Interaktion zwischen einer bestimmten Art von GUI-Elementen und dem Programm übernehmen:

- Ein *Selektor* entspricht einem einfachen Regulator. Er wird typischerweise bei Checkbuttons oder Radiobuttons eingesetzt, die abhängig vom Wert des Selektors automatisch z. B. zwischen den Zuständen *enabled* und *disabled* wechseln.
- Ein *Texteditor* erlaubt die Steuerung von ein- und mehrzeiligen Textfenstern und stellt dem Nutzer die Funktionalität eines einfachen Texteditors zur Verfügung.
- Ein *Canvas-Editor* wird zur Steuerung eines komplexen Zeichenfensters genutzt.
- Ein *Scroller* wird als Mediator zwischen Scrollbar und Canvas- oder Textfenster genutzt und ermöglicht so eine Anpassung des sichtbaren Bereichs von Texten und Bildern.

Man kann sich zahlreiche weitere Arten von Gates vorstellen. Wem es hier an Phantasie mangelt, der möge sich in der SWING Bibliothek von JAVA oder der WINDOWS FORMS Bibliothek von .NET Anregungen holen.

20.4.5 Ereignisse – Events

Wie wir schon beim Emitter gesehen haben, können Gates statt mit Graphik-Elementen auch mit deren ***Events (Ereignissen)*** verbunden werden. Ein Event ist dabei eine Interaktion mit dem Programm, die entweder durch den Benutzer oder durch die Umgebung hervorgerufen werden kann.

Die Struktur *Event* in Programm 20.11 stellt die Ereignisse bereit. Wir unterscheiden hier beispielsweise *Tasten-Ereignisse* wie *someKeyPress* oder *keyRelease*, *Mouse-Ereignisse* wie *focusIn* und *focusOut*, *Mousetasten-Ereignisse* wie *buttonPress* oder *buttonRelease* und *Bewegungs-Ereignisse* wie *motion*.

Programm 20.11 Die Struktur *Event*

```
STRUCTURE Event

  TYPE Event = { configure, expose, destroy, enter, leave,  ... }
  TYPE Modifier = {control, shift, lock, meta, alt, button, double,  ... }
  FUN _ + _: Modifier × Event → Event
  FUN _ + _: Event × Event → Event
  FUN button: Nat → Event                                   -- buttonPress
  FUN key: Denotation → Event                               -- keyPress
```

Ereignisse werden aus Basisereignissen wie *key*("*A*") zu Sequenzen zusammengesetzt, z. B. *key*(" / ") + *key*("*a*"). Auf Basisereignisse können so genannte Modifikatoren (*Modifier*) angewendet werden, wie z. B. *meta* + *shift*. Bei Button-Events spezifiziert die Zahl die Tastennummer, bei Key-Events spezifiziert der Text den Namen des Tastensymbols.

20.5 HAGGIS, FUDGETS, FRANTK und andere

Auf dem Gebiet der funktionalen GUI-Gestaltung sind verschiedenste Ideen verfolgt worden. Wir wollen hier auf einige typische Systeme eingehen.

So wie OPALWIN auf dem Agenten-System von OPAL aufgebaut ist, nutzt HAGGIS [51] das Prozess-System von CONCURRENT HASKELL (vgl. Kapitel 19.7). Beide Systeme basieren grundsätzlich auf der Idee von nebenläufigen monadischen Agenten bzw. Prozessen. Wie beim OPALWIN-System werden User-Interface und Anwendung bei HAGGIS weitgehend getrennt betrachtet und das User-Interface kann durch so genannte Layout-Kombinatoren aus Interface-Komponenten zusammengesetzt werden.

FUDGETS [31] („Fudget“ steht für „*fu*nctional wi*dget*“ bzw. für „*fu*nctional window ga*dget*“) ist ebenfalls ein GUI-Toolkit für HASKELL; es basiert allerdings auf dem Prinzip der Ströme. Ein Fudget stellt im Wesentlichen einen Prozess dar, der durch Nachrichtenaustausch über Streams bzw. Kanäle mit anderen Fudgets kommuniziert. Ein Fudget wird dabei von zwei Streams „durchlaufen“: So genannte „High level“-Nachrichten (z. B. vom User eingegebene Daten) werden zum Informationsaustausch zwischen den Programmteilen der eigentlichen Anwendung genutzt, während „Low level“-Nachrichten, wie Events, nur durch das Fudget selbst verarbeitet werden. Fudgets lassen sich wieder zu komplexeren Fudgets komponieren und somit komplexe User-Interfaces aus einfachen GUI-Elementen zusammensetzen.

Das Graphik-System von CLEAN [10, 11] realisiert die Ein-/Ausgabe-Effekte über „gesteuerte“ Seiteneffekte. Ein abstrakter Wert, der den Zustand der Welt (oder Teile davon) repräsentiert, wird dabei single-threaded als zusätzlicher Parameter explizit durchs Programm gereicht. Diese Idee ähnelt

damit dem monadischen Ansatz, verlangt aber die explizite Handhabung des Zustands, wodurch die Programme weniger elegant werden.

Die HASKELL-Bibliothek FRANTK [125] benutzt die Konzepte Behaviour und Event des FRAN-Modells (Functional Reactive Animation [46]) zur Modellierung von Systemen in Abhängigkeit von der Zeit. Ein Event beschreibt dabei einen diskret auftretenden Wert, wie z. B. einen Button-Klick, und ein Behaviour ein kontinuierliches Verhalten. Die Interaktion dieser Konzepte erlaubt dann die Änderung des Verhaltens eines Systems durch Events.

Beim GEC-System [9] werden aus Daten so genannte GECs (Graphical Editor Component) gebildet. Eine GEC ordnet einem Datenelement anhand des Aufbaus des entsprechenden Datentyps Darstellungseigenschaften zu (man nutzt hier die so genannte generische Programmierung) und legt Abhängigkeiten zwischen Daten, und damit von anderen GECs, fest. Die Darstellung kann vom Benutzer angepasst und die einzelnen Komponenten zu einer GUI kombiniert werden. Über die gegenseitigen Abhängigkeiten der Daten wird die GUI-Funktionalität realisiert.

Alle bisher genannten Bibliotheken und Systeme beschreiben GUIs auf recht abstraktem Niveau und versuchen, die GUI-Konstrukte deklarativ in die funktionale Sprache einzubetten. Demgegenüber stellen Bibliotheken wie TCLHASKELL [7], WXHASKELL [92], GTK+HS [2], GTK2HS [4] Komponenten und Funktionalität von TCL/TK, WXWIDGETS oder GTK+ meist über ein monadisches Interface, aber im Stil trotzdem häufig eher imperativ, in HASKELL bereit. Einige der Higher-Level-Systeme sind auf der Basis der Lower-Level-Bibliotheken implementiert, beispielsweise beruht FRANTK auf TCLHASKELL.

21

Massiv parallele Programme

Die Vielzahl ist ein Zaubermittel, das wir brauchen dürfen, um den Rhythmus zu schaffen, das aber alles verdirbt, wo wir sie gedankenlos wuchern lassen.
Hugo von Hofmannsthal
Wir könnten viel, wenn wir zusammenstünden.
Friedrich Schiller (Wilhelm Tell)

Die *Parallelisierung* von Programmen verspricht gegenüber der herkömmlichen sequenziellen Abarbeitung eine Leistungssteigerung und ermöglicht so eine effizientere Behandlung von rechenintensiven Problemen, wie sie z. B. in der Strömungsmechanik, der Bildverarbeitung, der Signalanalyse, bei der Wettervorhersage und bei vielen anderen Simulations- und Modellberechnungen auftreten.

Neben den bekannten parallelen Implementierungen für imperative Sprachen, wie HIGH PERFORMANCE FORTRAN (HPF) oder den Bibliotheken PVM und MPI zur Programmierung nach dem *Message Passing* Modell in C, C++ und FORTRAN, gibt es auch für funktionale Sprachen verschiedene Ansätze zur Parallelisierung.

Zunächst ist in funktionalen Ausdrücken schon eine *implizite Parallelität* vorhanden, denn im Allgemeinen ist bei einem Funktionsaufruf $f(e_1, ..., e_n)$ die Reihenfolge, in der die Argumente e_1 bis e_n ausgewertet werden, nicht festgelegt. Man kann sogar versuchen, parallel zur Auswertung der Argumente auch den Funktionsaufruf selbst auszuwerten. Zwar muss bei dieser Form der Parallelität, die man *Ausdrucksparallelität* nennt, vom Benutzer keine intellektuelle oder schreibtechnische Arbeit in die Parallelisierung investiert werden, aber es bedeutet auch, dass wichtige Effizienzfaktoren nicht adäquat berücksichtigt werden können; dazu gehören insbesondere die Granularität und die Lastverteilung. Diese müssen dann durch Heuristiken bestimmt werden, durch die aber deutliche Performance-Steigerungen kaum erreichbar sind (vgl. [89]).

Durch die Einführung von Metaanweisungen, die Daten- und Lastverteilung sowie Kommunikation *explizit* regeln (wie in den meisten imperativen

Sprachen), wird der Programmierer wieder mit diesen Aufgaben belastet, und die Eleganz und Abstraktheit der funktionalen Programme leiden.

Daher wird seit einigen Jahren intensiv ein spezielles Konzept untersucht: *Algorithmische Skelette* sind ein Kompromiss zwischen den Extremen der expliziten imperativen Parallelprogrammierung und der impliziten funktionalen Ausdrucksparallelität. In diesem Kapitel stellen wir die Programmierung massiv paralleler Systeme mit Hilfe solcher Skelette vor.

Massiv parallele Systeme bestehen aus einer großen Anzahl von Prozessoren, denen jeweils eigener lokaler Speicher zugeordnet ist und die durch Nachrichten miteinander kommunizieren. Das heißt, wir gehen hier von einer MIMD-Parallelrechnerarchitektur (Multiple Instruction Multiple Data) aus. Dabei beschränken wir uns auf den SPMD-Programmierstil (Single Program Multiple Data), bei dem das gleiche Programm auf allen Prozessoren gleichzeitig ausgeführt wird.

21.1 *Skeletons*: Parallelität durch spezielle Funktionale

Algorithmische Skelette [36] ermöglichen parallele Programmierung auf einem hohen Abstraktionsniveau. Skelette repräsentieren typische Parallelisierungsmuster, wie Farm, Map, Reduce, Branch&Bound und andere Baumsuchverfahren sowie Divide&Conquer und können vom Nutzer einfach für seine jeweilige Anwendung instanziiert werden.

In funktionalen Sprachen wird diese Idee durch spezielle Funktionen höherer Ordnung realisiert, die eine effiziente parallele Implementierung besitzen. Diese Skelette werden dabei in ansonsten sequenzielle Sprachen eingebettet und sind dann meist die einzige Quelle von Parallelität im Programm. Parallele Implementierungsdetails sind innerhalb der Skelette verborgen. Auf diese Weise kann man die Eleganz der Funktionalen Programmierung mit der Effizienz von Algorithmen für spezielle Anwendungsmuster verbinden.

Als Beispiel betrachten wir die Map-Funktion $*$ auf Sequenzen. In der sequenziellen Version durchläuft Map eine Liste schrittweise von vorn nach hinten und wendet die Argumentfunktion jeweils auf das aktuelle Listenelement an.

$$\text{FUN } \_ * \_ : (\alpha \to \beta) \times Seq\ \alpha \to Seq\ \beta$$
$$\text{DEF } f * \Diamond = \Diamond$$
$$\text{DEF } f * (a\ \cdot\!: rest) = (f\ \ a)\ \cdot\!: (f * rest)$$

Da das Ergebnis der Anwendung von f auf ein Listenelement jeweils völlig unabhängig von allen anderen Listenelementen ist, kann man hier in einfacher Weise parallelisieren: Man teilt die Liste einfach in Teillisten auf und wendet auf diese parallel die ursprüngliche sequenzielle Map-Funktion an (s. Abbildung 21.1). Das entspricht dem Parallelisierungsmuster *Farm*. Dieses Vorgehen kann man entsprechend auf andere Datenstrukturen wie Arrays oder Bäume übertragen.

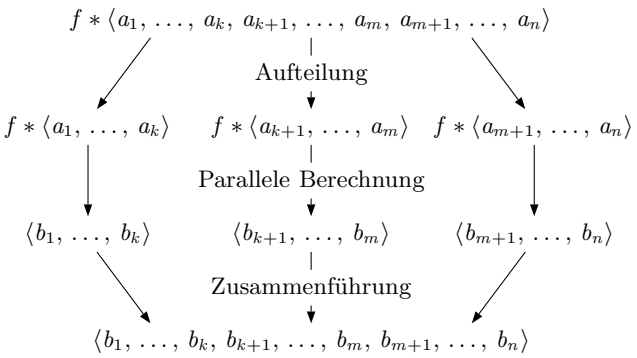


Abb. 21.1: Paralleles Map auf einer verteilten Liste

Für das parallele Map müssen die Listen nun aber auf eine andere Art implementiert sein als bisher. Mit der herkömmlichen induktiven Listendefinition würden wir eine Liste gleich drei Mal (zur Aufteilung, bei der Berechnung und beim Zusammensetzen des Resultats) durchlaufen, statt nur einmal wie beim sequenziellen Map. Das bedeutet, wir müssen dem Nutzer neben den Skeletten passende Datenstrukturen zur Verfügung stellen, die den Verteilungsaspekt berücksichtigen.

Skelette, die auf verteilten Datenstrukturen arbeiten, nennt man *datenparallele* Skelette. Hierbei geht man im Allgemeinen davon aus, dass die zu verarbeitenden Daten von Anfang an auf mehrere Prozessoren verteilt vorliegen. *Datenparallelität* ist in der Literatur vielfältig untersucht worden und auch wir werden uns im Folgenden damit beschäftigen.[1]

Die Implementierung der parallelen Map-Funktion als Skelett haben wir hier ausgelassen. Für den Nutzer ist sie in vielen Sprachen tatsächlich auch verborgen (PMLS [94], DPFL [89]); ihm steht dann üblicherweise eine Bibliothek vordefinierter Skelette zur Verfügung. Einige Sprachen, z. B. Eden [25] (vgl. Abschnitt 19.7), erlauben aber auch, parallele Skelette selbst zu implementieren.

Die Programmierung einer parallelen Anwendung mit algorithmischen Skeletten erfordert folgende Schritte:

- Erkennung der dem Problem inhärenten Parallelität,
- Auswahl adäquater Datenverteilungen (Granularität),
- Auswahl passender Skelette aus der Bibliothek und
- problemspezifische Instanziierung der Skelette.

[1] Im Gegensatz dazu spricht man von *Taskparallelität* (bzw. *control-oriented parallelism* [140]), wenn die zu verteilenden Prozesse und Daten nicht von vornherein bekannt sind und dynamisch erzeugt werden.

Wir wollen uns im Folgenden mit der Strukturierung bzw. Verteilung der Daten beschäftigen, denn sie ist ausschlaggebend für die gesamte Struktur des Algorithmus.

21.2 *Cover*: Aufteilung des Datenraums

Beim Design eines parallelen Algorithmus spielt neben der Auswahl passender Skelette die Verteilung der Daten auf die vorhandenen Prozessoren eine entscheidende Rolle.

Einerseits möchte man den Datenaustausch zwischen den Prozessoren gering halten, andererseits will man aber von der möglichen Parallelität profitieren. Wenn, wie beim Map-Skelett, die Operationen völlig unabhängig auf jedem Datenelement einzeln ausgeführt werden, ist eine Verteilung der Daten problemlos. Oft lassen sich auch Teilmengen von zusammenhängenden Datenelementen finden, so dass die einzelnen Elemente einer Teilmenge zwar untereinander abhängig, die Teilmengen insgesamt jedoch weitgehend voneinander unabhängig sind. Wirklich kompliziert wird die Aufteilung der Daten, wenn starke Abhängigkeiten bestehen oder wenn in unterschiedlichen Teilschritten der Berechnung unterschiedliche Aufteilungen von Vorteil wären. Ein solches Beispiel betrachten wir in Kapitel 21.3.

Wir stellen im Folgenden das Konzept der *Cover* [114, 137, 104, 105] vor, die sowohl die *Datenverteilung* als auch die *Kommunikation* zwischen den Prozessoren realisieren.

Anmerkung: Im Gegensatz zu unseren Covern trennt beispielsweise [89] diese Konzepte in Datenverteilung zum Zeitpunkt der Erzeugung der verteilten Datenstrukturen und explizite Kommunikationsoperationen, so genannte Kommunikationsskelette. Beide Ansätze verzichten bei der Kommunikation aber auf explizite send-/receive-Anweisungen und entkoppeln die Datenaufteilung und Kommunikation von den eigentlichen Berechnungsaufgaben.

Cover beschreiben die Zerlegung und die Kommunikationsmuster einer Datenstruktur. Eine einfache Zerlegung einer Liste, wie sie das parallele Map im vorangehenden Abschnitt verwendet, zeigt Abbildung 21.2. Die Liste wird hier auf die drei parallel arbeitenden Prozessoren p_0, p_1 und p_2 verteilt.

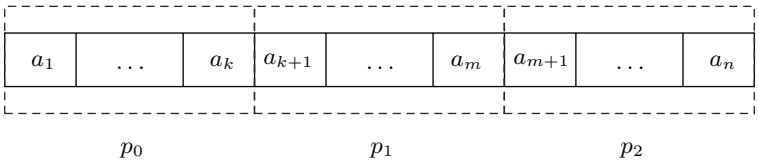


Abb. 21.2: Ein einfaches Listen-Cover

Partitionen verteilter Datenstrukturen können sich *überlappen*. Das bedeutet, dass ein Prozessor zusätzlich zur eigenen Teildatenstruktur auch Elemente der Nachbarpartitionen sieht. Auf diese Weise können die Prozessoren miteinander kommunizieren. Während ein Prozessor die Elemente seines eigenen Bereichs aber ändern kann, gilt das für die Kopien der Nachbarpartitionen nicht. Diese sind für ihn lediglich sichtbar, um Lesezugriffe direkt zu ermöglichen und somit eine explizite Programmierung der Kommunikation zu vermeiden.

Abbildung 21.3 zeigt ein Listen-Cover mit *Überlappungen*. Neben vier eigenen Elementen je Prozessor überlappen sich die Teildatenstrukturen um je ein Element nach links bzw. rechts. Die inneren vier (dunklen) Elemente stellen die eigene Subliste des Prozessors p_i dar, der lesend außerdem auf je ein Element der Prozessoren p_{i-1} und p_{i+1} zugreifen kann. Bei einer realen Implementierung wird man natürlich sehr viel größere Teildatenstrukturen wählen, um ein sinnvolles Verhältnis von Kommunikation und Berechnungaufwand sicherzustellen.

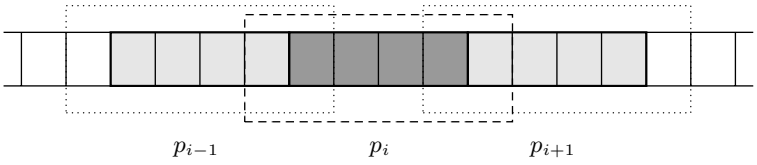


Abb. 21.3: Ein Listen-Cover mit überlappenden Elementen

Bei Veränderung sich überlappender Daten muss zu passender Zeit für ein entsprechendes Update gesorgt werden. Diese Daten können dann meist als Block übertragen werden, was effizienter ist, als die Daten immer dann, wenn sie gebraucht werden, einzeln zu übertragen.

21.2.1 Spezifikation von Covern

Wenn wir konkrete Cover definieren, dann geben wir sie als eine Instanz des allgemeinen Schemas in Programm 21.1 an.
Dabei bezeichnet $S\ \alpha$ den Typ des durch das Cover zerlegten Objekts, $C\ \beta$ den Typ des Covers und $U\ \gamma$ den Typ der Subobjekte. Die Funktionen *split* und *glue* definieren die Beziehung zwischen Objekt und Cover: Mit *split* zerlegen wir das Originalobjekt entsprechend des Covers in Subobjekte, und *glue* fügt die Subobjekte wieder zusammen.

Je nachdem, wie wir $S\ \alpha$, $C\ \beta$ und $U\ \gamma$ wählen, können wir Listen, Matrizen, Bäume, Graphen und andere Datenstrukturen aufteilen. Und für jede Datenstruktur kann man – je nach Algorithmendesign – auch verschiedene Cover angeben (und diese sogar gemeinsam innerhalb eines Algorithmus nutzen).

Programm 21.1 Allgemeine Cover-Struktur

```
COVER Cover = {
  TYPE S α                    -- das Gesamtobjekt
       C β                    -- das Cover
       U γ                    -- die lokalen Subobjekte
  FUN split: S α → C (U α)    -- Zerlegen des Originalobjekts
  FUN glue: C (U α) → S α     -- Zusammenfügen des Originalobjekts
  PROP glue ∘ split = id
}
```

Eine verteilte Liste wie in Abbildung 21.2 können wir als Sequenz von Listen (vgl. Programm 21.2) darstellen.

Programm 21.2 Definition eines einfachen Listen-Covers

```
COVER SeqBySeq[p] = {
  EXTEND Cover RENAMING (S, C, U) AS (Seq, Seq, Seq)
  FUN p: nat                                  -- Anzahl der Prozessoren/Subobjekte
  FUN split: Seq α → Seq (Seq α)
  DEF split s = LET size = ⌈(length s) / p⌉ -- Größe der Teillisten
                IN   split' s size
  FUN split': Seq α → Nat → Seq (Seq α)
  DEF split's size = IF length s = size
                     THEN ⟨s⟩
                     ELSE take(size, s) ·: split'(drop(size, s))(size)
                FI
  FUN glue: Seq (Seq α) → Seq α
  DEF glue = ++ /
}
```

Wenn wir Überlappungen wie in Abbildung 21.3 realisieren wollen, müssen wir das ebenfalls in der Cover-Beschreibung berücksichtigen. In Programm 21.3 haben wir eine solche Cover-Definition angegeben. Die beiden zusätzlichen Schlüsselwörter OWN und FOREIGN werden zur Beschreibung von Überlappungen verwendet. Der OWN-Teil des Datentyps *SubSeq* bezeichnet die „eigenen“ Elemente des jeweiligen Prozessors, d. h. die Elemente, auf die lesend und schreibend zugegriffen werden kann. Die mit FOREIGN gekennzeichneten Elemente sind Teildatenstrukturen anderer Prozessoren, die der lokale Prozessor zwar lesen, aber nicht verändern kann.

In unseren Cover-Definitionen sind die Funktionen *split* und *glue* ganz gewöhnliche Funktionen. Durch *split* zerlegen wir eine Datenstruktur in Teildatenstrukturen. In der unterliegenden Implementierung werden diese dann

Programm 21.3 Definition eines Listen-Covers mit Überlappungen

```
COVER SeqBySubSeq[l, r, p] = {
  EXTEND Cover RENAMING (S, C, U) AS (Seq, Seq, SubSeq)
  FUN l, r: nat                                  -- Größe der FOREIGN-Anteile
  FUN p: nat                                     -- Anzahl der Prozessoren/Subobjekte
  TYPE SubSeq = (FOREIGN left  : Seq,
                 OWN     inner : Seq,
                 FOREIGN right : Seq )
  FUN glue: Seq (SubSeq α) → Seq α
  DEF glue s = ++ / s | ◊
  FUN ++: SubSeq α × Seq α → Seq α
  DEF (l, i, r) ++ seq = l ++ i ++ r ++ seq
  PROP ∀(left, inner, right) ∈ (split s) • (length left) = l ∧ (length right) = r
  PROP length(split s) = p
       ...
}
```

aber Teiltasks zugeordnet und tatsächlich parallel bearbeitet. Entsprechend sammelt *glue* die Ergebnisse der Teilberechnungen wieder zusammen und bildet daraus die neue Gesamtdatenstruktur. Dies kann in vielen Fällen sogar vom Compiler automatisch effizient umgesetzt werden [105].

Anmerkung 1: Cover-Definitionen kann man zu so genannten Data Distribution Algebras [114, 137] geeignet zusammenfassen. Eine solche Algebra umfasst dann eine Menge von Datenstruktur-Zerlegungen, die bzgl. der Transformation zwischen den Verteilungen abgeschlossen ist und die Ableitung von parallelen Algorithmen durch Programmtransformation erlaubt.

Anmerkung 2: Bei vielen Algorithmen werden die Datenstrukturen inkrementell verändert; diesen Effekt haben wir schon in Kapitel 10 im Zusammenhang mit den so genannten Mikroschritten kennengelernt. In der parallelen Realisierung solcher Programme heißt dies, dass man z. B. bei Matrixalgorithmen teilweise noch auf Werte der „alten" und teilweise schon auf Werte der „neuen" Matrix zugreift. Das lässt sich mit einem speziellen Schlüsselwort wie FUTURE *kennzeichnen und systematisch in das funktionale Konzept einfügen [114, 137].*

21.2.2 Skelette über Covern

Basierend auf Cover-Definitionen kann man für typische Parallelisierungsmuster Skelette als Funktionen höherer Ordnung implementieren. Die wichtigsten dieser Skelette werden sinnvollerweise vordefiniert und in Bibliotheken zusammengefasst. Der Programmierer braucht aber auch die Möglichkeit, selbst Skelette zu definieren. Beide Arten von Skelettdefinitionen können sowohl für spezielle Cover-Instanzen als auch für das allgemeine Cover-Schema erfolgen.

Programm 21.4 zeigt die Definition von Skeletten über dem allgemeinen Cover-Schema für das Map-Filter-Reduce-Paradigma.

Programm 21.4 *Map-Reduce* auf Covern

```
SKELETONS HigherOrder OVER Cover = {
  FUN _ * : (U α → U β) → S α → S β
  DEF g* = glue ∘ (g*) ∘ split
  FUN ⋁ : (U α × U α → U β) → (S α × S α → S β)
  DEF a ⋁^g b = glue ∘ (⋁^g)(split a, split b)
  FUN _ / _ | _ : (U α × β → β) × S α × β → β
  DEF g / s | e = (g / _ | e) ∘ split s
}
```

Die Map-Funktion $*$ zerlegt eine gegebene Datenstruktur s, wendet dann die Funktion g auf alle Subobjekte von s an und fügt die Ergebnisse wieder zusammen. Dabei wird eine Map-Funktion $*$ für den Cover-Typ $C\gamma$ vorausgesetzt.

Ähnlich verhalten sich die Zip- und die Reduce-Funktion; auch hier setzten wir die entsprechenden Funktionen höherer Ordnung auf dem Cover-Datentyp voraus. Bei Reduce (g muss hier assoziativ sein) werden die Einzelergebnisse mit Hilfe der Reduce-Funktion auf dem Cover-Typ zusammengefasst, daher entfällt hier die Anwendung von *glue*.

Eine Filter-Funktion haben wir nicht angegeben, da Filtern nicht nur die Werte, sondern im Allgemeinen auch die Datenstruktur ändert. Ihre Anwendung wäre daher hier nur in bestimmten Fällen sinnvoll.

Anmerkung: Alternativ hätte man übrigens Map, Zip und Reduce auch für Funktionen, die direkt auf den Datenelementen der Gesamtstruktur arbeiten, definieren können. Für die Map-Funktion sähe das beispielsweise so aus:

```
FUN _ * : (α → β) → S α → S β
DEF g* = glue ∘ ((g*)*) ∘ split
```

Dabei setzen wir dann je eine Map-Funktion $*$ für den Cover-Typ und den Typ der Cover-Elemente voraus.

Wie man an diesen Definitionen sieht, würde z. B. bei der Komposition mehrerer Map-Funktionen $(h*) \circ (g*) \circ (f*)$ die Datenstruktur zwischen je zwei Anwendungen mit *glue* zusammengebaut und sofort wieder mit dem nächsten *split* zerlegt und verteilt werden. Solche Ineffizienten müssen durch die Optimierung des Compilers gefunden werden [105]. Das ist offensichtlich eine nichttriviale Aufgabe.

21.2.3 Matrix-Cover

Eine in der datenparallelen Programmierung oft verwendete Datenstruktur sind Matrizen (Arrays, vgl. Kapitel 14). Eine Matrix kann man z. B. als Folge von Zeilen verteilt darstellen (s. Programm 21.5). Genausogut können wir eine

Programm 21.5 Matrix als Folge von Zeilen

```
COVER MByRows = {
  EXTEND Cover RENAMING (S, C, U) AS (Matrix, Seq, Seq)
  PROP split m = «Matrix m als Sequenz von Zeilen»
       ...
}
```

Matrix durch eine Sequenz von Spalten oder als eine Matrix von Matrizen, mit oder ohne Überlagerungen wählen, wie es z. B. in Programm 21.6 gezeigt ist.

Programm 21.6 Matrix als Matrix von Matrizen (mit Überlagerung)

```
COVER MByM[m, n, p] = {
  EXTEND Cover RENAMING (S, C, U) AS
                        (Matrix[m, m], Matrix[p, p], SubMatrix[n, n])
  FUN m, n, p: nat
  TYPE SubMatrix[n, n] = ( OWN     left  : Matrix[n, n],
                           FOREIGN right : Matrix[n, n] )
  PROP split mx = «m × m-Matrix mx als p × p-Matrix von n × n-überlappenden
                   2n × n-Submatrizen, mit n = m / p»
                   ...
}
```

Die Spezifikation *MByM* fixiert ein Cover auf $m \times m$-Matrizen in Form einer $p \times p$-Matrix. Jedes der Subobjekte ist selbst eine $2n \times n$-Matrix (mit $n = m/p$), die zur (linken) Hälfte dem eigenen Prozessor „gehört“ und zur (rechten) Hälfte einem Nachbarprozessor.

Abbildung 21.4 skizziert eine solche Zerlegung einer Matrix A. Die Prozessoren, denen die Cover-Elemente an den Positionen $A_{i,p}$, $i \in \{1, \ldots, p\}$, d. h. am „rechten“ Rand des Covers, zugeordnet sind, dürfen außerdem lesend auf das jeweilige Cover-Element $A_{i,1}$ am „linken“ Rand zugreifen. Eine solche Verteilung lässt sich einfach auf eine Torusarchitektur (vgl. Abbildung 21.5) mappen. Im folgenden Abschnitt betrachten wir ein Beispiel zur Programmierung mit Skeletten und Covern über einer solchen Datenverteilung.

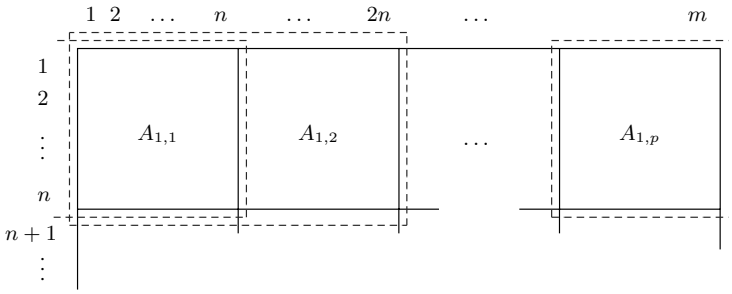


Abb. 21.4: Zerlegung einer Matrix entsprechend des *MByM*-Covers

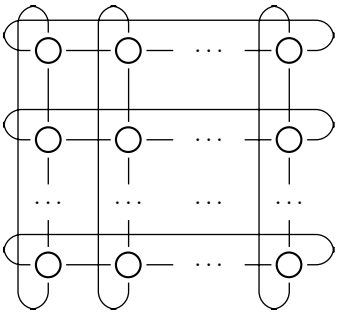


Abb. 21.5: Torusarchitektur über Prozessoren

21.3 Beispiel: Matrixmultiplikation

Wir wollen ein Programm zur parallelen Multiplikation $A \cdot B$ zweier $m \times m$-Matrizen A und B auf einer Torusarchitektur von $p \times p$ Prozessoren nach Gentleman [56] entwickeln. Während dort auf jedem der Prozessoren jeweils genau ein Element der Matrizen A und B gehalten werden, gehen wir – wie in [89] gezeigt – davon aus, dass stattdessen auf jedem Prozessor je eine $n \times n$-Submatrix von A und B mit $n = m/p$ verwaltet wird. Dabei nehmen wir an, dass wir die $m \times m$-Matrizen sauber aufteilen können, d. h., es gilt $m \; mod \; p = 0$.

Zunächst müssen wir also unsere $m \times m$-Matrizen A und B in $p \times p$-Matrizen über $n \times n$-Submatrizen zerlegen, wie es Abbildung 21.6 für die Matrix A über den Elementen $a_{i,j}$ zeigt.

Diese Zerlegung gibt uns schon grob die notwendige Cover-Struktur vor. Bevor wir aber eine passende Cover-Definition angeben können, müssen wir den Algorithmus hinsichtlich des Datenaustauschs zwischen den Prozessoren analysieren, denn das Cover legt mit Hilfe der Überlappungsbereiche auch die Kommunikationsstruktur fest.

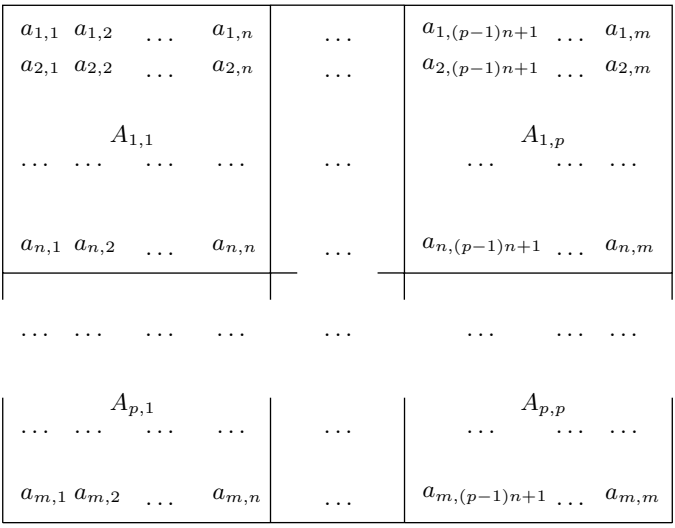


Abb. 21.6: Zerlegung einer $m \times m$-Matrix in $p \times p$ $n \times n$-Submatrizen

Sehen wir uns also zunächst den Algorithmus genauer an: Auf jedem der Prozessoren ist je eine $n \times n$-Submatrix von A und B lokal verfügbar. Daneben verwaltet jeder Prozessor lokal eine $n \times n$-Submatrix als Zwischenergebnis der späteren $m \times m$-Resultatmatrix C. Nach einer *Initialisierung* (die wir weiter unten beschreiben werden) durchläuft der Algorithmus p-mal die folgenden drei Schritte:

1. Auf jedem der $p \times p$ Prozessoren werden die jeweiligen Submatrizen von A und B lokal zu einer neuen $n \times n$-Matrix D multipliziert.
2. Dieses lokale Ergebnis D wird zum bisherigen lokalen Zwischenergebnis C' hinzuaddiert.
3. Damit alle notwendigen Multiplikationen von Submatrizen ausgeführt werden können, müssen danach die Elemente der $p \times p$-Matrizen über der Menge der Prozessoren rotieren. Dabei müssen die Submatrizen von A schrittweise nach links und die Submatrizen von B schrittweise nach oben verschoben werden.

Für Schritt (1) benötigen wir eine Multiplikations-Funktion *mul* auf $n \times n$-Matrizen, die die Multiplikation lokal (und damit sequenziell) auf dem jeweiligen Prozessor durchführt. In Schritt (2) addieren wir, ebenfalls lokal, zwei $n \times n$-Matrizen. Beides wurde im Wesentlichen schon in Kapitel 14 behandelt, ist aber zur besseren Lesbarkeit in Programm 21.7 noch einmal gezeigt.

Im Gegensatz zu *add* und *mul*, die lokal auf den einzelnen Prozessoren durchgeführt werden, stellt die Verschiebung von Submatrizen in Schritt (3) eine Kommunikation zwischen den Prozessoren dar.

Programm 21.7 Lokale Matrizenmultiplikation und -addition

```
FUN mul: Matrix[n, n] × Matrix[n, n] → Matrix[n, n]
PROP mul (A, B) = D ⟺ ∀i, j • D_{i,j} = skalProd (row(A, i), col(B, j))
FUN skalProd: Row × Column → Real
DEF skalProd (a, b) = + / (· ∗ (a, b))
FUN add: Matrix[n, n] × Matrix[n, n] → Matrix[n, n]
PROP add (C', D) = C ⟺ ∀i, j • C_{i,j} = C'_{i,j} + D_{i,j}
```

Da Kommunikationen hier indirekt über die Cover geregelt werden, müssen wir unser Cover so wählen, dass wir die Rotationen lokal beschreiben können. Das bedeutet, dass wir neben der lokalen $n \times n$-Matrix (im OWN-Part) zusätzlich die rechts (für Matrix A) bzw. die unten (für Matrix B) benachbarte $n \times n$-Matrix als FOREIGN-Part bereithalten müssen. Eine solche Cover-Definition *MByM* für Matrix A haben wir schon in Programm 21.6 angegeben. Für Matrix B gehen wir analog vor (vgl. Programm 21.8).

Programm 21.8 Matrix-Cover (mit Überlagerung nach „unten“)

```
COVER MByM'[m, n, p] = {
  EXTEND Cover RENAMING (S, C, U) AS
                        (Matrix[m, m], Matrix[p, p], SubMatrix'[n, n])
  FUN m, n, p: nat
  TYPE SubMatrix'[n, n] = ( OWN     up   : Matrix[n, n],
                            FOREIGN down : Matrix[n, n] )
  PROP split' mx = «m × m-Matrix mx als n × 2n-Submatrizen»
                   ...
}
```

Nun benötigen wir noch die Rotationsfunktionen. Diese arbeiten auf den Subobjekten gemäß der Matrix-Cover *MByM* und *MByM'* und verschieben bzw. kopieren den jeweiligen FOREIGN-Part in den OWN-Bereich. Dies ist in Programm 21.9 angegeben. Werden diese Funktionen (gleichzeitig für alle Prozessoren) aufgerufen, lösen sie eine Kommunikation zwischen den Prozessoren aus.

Jetzt müssen wir noch die schon genannte Initialisierung der Matrizen vorgeben. Diese beruht ebenfalls auf horizontaler bzw. vertikaler Rotation von A und B mittels *shiftRow* und *shiftCol*. Dabei wird die i-te Zeile der $p \times p$-Matrix A um $(i-1)$ Schritte nach links und die j-te Spalte der $p \times p$-Matrix B um $(j-1)$ Schritte nach oben verschoben. Das führt zu den in Abbildung 21.7 angegebenen Matrizen A' und B'.

Programm 21.9 Rotation von Matrizen

```
FUN shiftRow: SubMatrix[n, n] → SubMatrix[n, n]
DEF shiftRow (L, R) = (R, R)
FUN shiftCol: SubMatrix'[n, n] → SubMatrix'[n, n]
DEF shiftCol (U, D) = (D, D)
```

$A_{1,1}$	$A_{1,2}$	$\cdots$	$A_{1,p}$
$A_{2,2}$	$A_{2,3}$	$\cdots$	$A_{2,1}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$A_{p,p}$	$A_{p,1}$	$\cdots$	$A_{p,p-1}$

$B_{1,1}$	$B_{2,2}$	$\cdots$	$B_{p,p}$
$B_{2,1}$	$B_{3,2}$	$\cdots$	$B_{1,p}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$B_{p,1}$	$B_{1,2}$	$\cdots$	$B_{p-1,p}$

Abb. 21.7: Rotierte Submatrizen A' und B' nach der Initialisierung

Jetzt haben wir alle notwendigen Hilfsfunktionen definiert und können die parallele Multiplikation der $m \times m$-Matrizen betrachten. In Programm 21.10 haben wir die Implementierung angegeben.

Programm 21.10 Parallele Matrix-Multiplikation nach Gentleman

```
FUN matrixMultiplication: Matrix[m, m] × Matrix[m, m] → Matrix[m, m]
DEF matrixMultiplication (A, B) =
    LET A_dist  =  MByM.split(A)
        B_dist  =  MByM'.split'(B)
        C_dist  =  MByM''.split''(0_{m,m})
        (A'_dist, B'_dist) = initialize(A_dist, B_dist)
    IN  distMul(A'_dist, B'_dist, C_dist, p)
FUN initialize: Matrix[p, p] × Matrix[p, p] → Matrix[p, p] × Matrix[p, p]
PROP initialize(A, B) = (A', B') ⟺
     ∀i ∈ 1, ..., p • A'_{i,j} = shiftRow^i(A_{i,j})
     ∀j ∈ 1, ..., p • B'_{i,j} = shiftCol^j(B_{i,j})
FUN distMul: Matrix[p, p] × Matrix[p, p] × Matrix[p, p] × Nat → Matrix[m, m]
DEF distMul(A, B, C, 0) = MByM''.glue''(C)
DEF distMul(A, B, C, p) = LET C'  =  (left * A) ⨁^mul (up * B)      -- Schritt (1)
                              C'' =  C ⨁^add C'                     -- Schritt (2)
                              A'  =  shiftRow * A                   -- Schritt (3)
                              B'  =  shiftCol * B
                          IN  distMul (A', B', C'', (p − 1))
```

Zuerst initialisieren wir die Matrizen A und B und legen eine verteilte Matrix C_{dist} als Null-Matrix an, in der die Zwischenergebnisse aufaddiert werden. Die Cover-Definition $MByM''$ für die Matrix C ist in Programm 21.11 angegeben. Es handelt sich hierbei um eine einfache Zerlegung einer Matrix in Submatrizen ohne Überlappungen.

Der Funktionaufruf $distMul(A'_{dist}, B'_{dist}, C_{dist}, p)$ führt nun p-mal die Schritte (1), (2) und (3) des Algorithmus durch und fügt am Ende die $p \times p$ Submatrizen der Ergebnismatrix C zu einer $m \times m$-Matrix zusammen.

Programm 21.11 Ein einfaches Matrix-Cover

```
COVER MByM''[m, n, p] = {
  EXTEND Cover RENAMING (S, C, U) AS
                        (Matrix[m, m], Matrix[p, p], Matrix[n, n])
  FUN m, n, p: nat
  PROP split'' = «m × m-Matrix als p × p n × n-Submatrizen»
  DEF glue = ...
}
```

Betrachten wir unsere Lösung genauer, dann sehen wir, dass wir die Funktionen (abgesehen von den Cover-Funktionen *split* und *glue*) in vier Gruppen einteilen können:

- Die Funktionen *mul* und *add* arbeiten sequenziell und sind nur auf den lokalen Daten der Prozessoren (also auf $n \times n$-Matrizen) definiert.
- Ebenso sind die Funktionen *shiftRow* und *shiftCol* nur auf den jeweiligen lokalen Daten der Prozessoren definiert, allerdings greifen sie (lesend) auf die Datenkopien anderer Prozessoren zu und überschreiben eigene Daten, die auf anderen Prozessoren als Kopien verfügbar sind. Dadurch lösen sie Kommunikationen zwischen den Prozessoren aus.
- Zur dritten Gruppe zählen wir die Funktionen *matrixMultiplication*, *distMul* und *initialize*, die auf den globalen Matrizen zwar sequenziell arbeiten, aber parallel implementierte Skelette der vierten Gruppe aufrufen.
- Die vierte Gruppe umfasst parallele Skelette wie z. B. Map ($*$) und Zip ($\bigvee$).

21.4 Von Skeletons zum *Message passing*

Damit der Nutzer einer Skelett-Bibliothek bequem auf der Basis von Cover-Definitionen und vordefinierten (und eigenen) Skeletten effiziente parallele Programme schreiben kann, müssen die eigentliche parallele Skelettimplementierung bzw. die Umsetzung der Funktionen *shift* und *glue*, die Datenver-

teilung und die Kommunikation und Synchronisation sorgfältig auf unterer Ebene umgesetzt werden [105].

Die Übersetzung erfolgt auf der Basis einer Hierarchie von Skeletten, wobei abstraktere, anwendungsorientierte Skelette durch weniger abstrakte ersetzt werden, die das Programm mit Implementationsdetails wie z. B. speziellen Architektureigenschaften der Zielarchitektur verfeinern. Die Funktionen auf unterster Ebene können direkt mit Hilfe von Message-Passing-Bibliotheken wie PVM oder MPI für die parallele Zielarchitektur definiert werden.

Eine solche Skeletthierarchie erlaubt dabei unter Umständen verschiedene Transformationen, insbesondere da die (implizite) Spezifikation von Datenverteilung und Kommunikation basierend auf Cover-Definitionen (im Gegensatz zu direkten Kommunikationsskeletten wie in [89]) gewisse Freiheitsgrade lässt. Die Auswahl geeigneter Transformationsschritte wird dann mit Hilfe von Kostenfunktionen durchgeführt, die ebenfalls auf den Cover-Definitionen basieren [115].

22

Integration von Konzepten anderer Programmierparadigmen

Es ist aber schwer, die Natur einem Paradigma anzupassen.

Thomas S. Kuhn
(Die Struktur wissenschaftlicher Revolutionen)

In der Praxis hat man es häufig mit Aufgabenstellungen zu tun, bei denen sich zwar bestimmte Teilaufgaben sehr gut funktional beschreiben und lösen lassen, für die Programmierung anderer Teile des Gesamtsystems aber Konzepte anderer Programmierparadigmen geeigneter sind. Beispielsweise kann neben reinen Berechnungsaufgaben ein Teilsystem eine Datenbank sein und ein weiteres Modul die Benutzerinteraktion realisieren.

Durch die Integration von Konzepten unterschiedlicher Programmierparadigmen in einer Sprache ist es möglich, jeweils geeignete Sprachmittel zu verwenden, sodass jede Teillösung möglichst genau der Spezifikation entspricht. Dies unterstützt eine elegantere und klarere Programmierung, minimiert Fehlermöglichkeiten und erleichtert die Korrektheitsprüfung des Programms.

In diesem Kapitel betrachten wir die Integration von Konzepten anderer Programmierparadigmen in funktionale Sprachen.

22.1 Programmierparadigmen und deren Integration

Verschiedene Menschen haben unterschiedliche Sichtweisen bei der Beschreibung und Lösung von Problemen. Das hat sich auch in der Welt der Programmiersprachen durch verschiedene Programmierparadigmen manifestiert. Ein *Programmierparadigma* ist eine Sichtweise, die zur Lösung eines Problems mittels einer Programmiersprache eingenommen wird.

Funktionale Sprachen bauen auf dem Funktionsbegriff aus der Mathematik auf und eignen sich daher für Aufgaben, die sich direkt als Funktionen darstellen lassen. Das klingt zwar zunächst sehr eingeschränkt, aber wir haben schon gesehen, dass wir auf diese Weise eine ganze Reihe von Problemen,

wie z.B. Parsing, Approximationsaufgaben, Lösung von Gleichungssystemen und sogar Schedulingaufgaben, sehr elegant darstellen können. Für Suchaufgaben, Scheduling- und Optimierungsprobleme sowie für Design- und Diagnoseanwendungen eignen sich im Allgemeinen aber *logische und constraint-basierte Programmiersprachen* deutlich besser. Gegenüber diesen *deklarativen Sprachen* sind *imperative Sprachen* zweckmäßiger zur Modellierung von Abläufen in der realen Welt.

Sprachen unterscheiden sich also nicht nur in der Syntax, sondern auch in den Sprachmitteln und Konzepten, die sie zur Problemlösung bereitstellen. Dabei sind die Sprachmittel und oft auch die Syntax innerhalb eines Paradigmas wiederum (sehr) ähnlich. Zum Beispiel stehen in allen funktionalen Sprachen Konstrukte zur Definition, Applikation und Komposition von Funktionen sowie Funktionen höherer Ordnung zur Verfügung.

Zur Darstellung von Konzepten, die in anderen als dem funktionalen Paradigma typischer oder natürlicher sind, wie z. B. die Darstellung von Ein- und Ausgabe, gibt es oft verschiedene Ansätze zur Integration. Aber auch hier haben sich meist bestimmte bevorzugte Lösungen herauskristallisiert, wie für Ein- und Ausgabeoperationen beispielsweise Monaden (vgl. Kapitel 17).

Was wir bereits in früheren Kapiteln bei Objekten, Agenten und parallelen Skeletten – also jeweils für spezielle Aspekte – angeschnitten haben, wollen wir in diesem Kapitel genauer und allgemeiner betrachten. Es geht darum, mittels Paradigmenintegration Programmiermittel anderer Paradigmen für eine bequeme und effiziente Programmierung von komplexen Problemstellungen auch in funktionalen Sprachen bereitzustellen.

Wenn es sich nicht nur um eine Integration ausgewählter, einzelner Aspekte handelt, sondern um eine echte und umfassende Paradigmenintegration, so spricht man auch von *Multiparadigmen-Programmiersprachen*. Diese vereinigen die Ausdrucksmöglichkeiten mehrerer Programmierparadigmen, wie zum Beispiel der logischen, funktionalen oder objektorientierten Programmierung, in einer integrierten Programmiersprache. Eine ausführliche Betrachtung dazu findet man z. B. in [61].

Neben der klassischen Einteilung in deklarative und imperative Sprachen sind hier vor allem die Paradigmen der *objektorientierten, nebenläufigen und verteilten Programmierung* von Interesse. Hierbei klassifizieren wir nicht nach der Sichtweise der Problembeschreibung, sondern z. B. danach, wie einzelne Berechnungsschritte kombiniert werden, d. h. sequenziell oder nebenläufig bzw. verteilt. Diese Klassifikation verhält sich orthogonal zur vorher genannten, d. h., es existieren praktisch von allen Sprachparadigmen sowohl sequenzielle als auch nebenläufige Vertreter, oft auch verteilte. Und Ähnliches gilt auch für die objektorientierte Programmierung: Sowohl für imperative als auch für deklarative Programmiersprachen existieren entsprechende Erweiterungen.

22.2 Objektorientierte Erweiterungen funktionaler Sprachen

Es ist unstrittig, dass die objektorientierte Programmierung nicht nur ein vorübergehender Hype war (wie so vieles in der Informatik), sondern eine essenzielle Verbesserung der Programmierpraxis bewirkt hat. Zwar würden überzeugte funktionale Programmierer der objektorientierten Programmierung nie das Potenzial an Eleganz, Sicherheit und Produktivität zusprechen, das die Funktionale Programmierung auszeichnet, aber man muss doch zugeben, dass gewisse Dinge sich objektorientiert sehr gut ausdrücken lassen.

Wäre es unter diesen Umständen nicht schön, wenn man beide Paradigmen miteinander verbinden könnte?

Einen möglichen Lösungsansatz haben wir in Kapitel 18 schon eingehend diskutiert: Wenn man die Objekte von Sprachen wie C++, EIFFEL, JAVA oder C# ansieht, dann stellen sie Zugriffe auf Attributwerte und Methoden bereit, die sich im Lauf der Zeit ändern können. Deshalb haben wir sie über das Konzept der Zeit-Monade modelliert.

Aber das ist nicht die einzige Möglichkeit der Integration. In anderen Sprachen ist man leicht unterschiedliche Wege gegangen. Einige davon wollen wir im Folgenden wenigstens kurz skizzieren.

In Kapitel 9 haben wir das Konzept der Typklassen in funktionalen Sprachen betrachtet. Typklassen implementieren bereits Konzepte der objektorientierten Welt, wenn auch in sehr eingeschränkter Weise.

Typklassen systematisieren die Überladung von Funktionen in polymorphen Typsystemen. Eine Typklasse ist eine Menge von Typen zusammen mit zugehörigen Funktionen, die dabei für verschiedene konkrete Typen jeweils verschieden implementiert sein können. Vererbung zwischen Typklassen erlaubt es, Operationen, die schon in anderen Klassen definiert wurden, zu übernehmen (und diese neuen Klassen dann zu erweitern). Eine Klasse kann dabei auch mehrere Oberklassen haben.

Ähnlich wie bei der objektorientierten Programmierung realisieren Typklassen also Ideen wie das Zusammenfassen von Datenelementen bzw. Objekten mit ähnlichen Funktionen, deren Abstraktion sowie das Konzept der Vererbung. Die Vorteile objektorientierter Sprachen bei der Modellierung realer Systeme, die aus der Betonung der Objektstruktur beim Systemdesign und der Unterstützung von Interaktionen zwischen Komponenten entstehen, sind davon aber noch nicht berührt. Insbesondere zwei Eigenschaften von Objekten lassen sich in funktionalen Sprachen direkt zunächst nicht darstellen: Die Veränderbarkeit von Objekten, wie z. B. das Überschreiben von Attributwerten, und die Objektidentität.

Es gibt aber durchaus Sprachen, die echte Objektorientierung in die Funktionale Programmierung integrieren. Diese Sprachen unterscheiden sich dabei sowohl in der Herangehensweise und Technik der Integration als auch in den realisierten Konzepten.

22.2.1 HASKELL++

Die Sprache HASKELL++ [84] ist eine minimale Erweiterung des Typklassensystems von HASKELL um so genannte Objektklassen, deren wesentliches Ziel die Vererbung von Funktionen zwischen Instanzen derselben Objektklasse und damit bessere Codewiederverwendbarkeit war. HASKELL++-Programme werden nach HASKELL übersetzt. Objekte sind hierbei Werte abstrakter Datentypen; Funktionen (bzw. Methoden), die den Zustand eines Objekts verändern sollen, erzeugen letztendlich stattdessen ein neues Objekt.

22.2.2 O'HASKELL

O'HASKELL [106] ist eine Erweiterung von HASKELL um reaktive Objekte für nebenläufige Anwendungen. Hierbei werden Monaden verwendet, um Objekte mit Zuständen zu implementieren.

Zur Implementierung von Templates (diese sind das Äquivalent für Klassen in O'HASKELL) benutzt man Records. Das folgende Beispiel (angelehnt an [106]) zeigt einen Recordtyp, der einen Punkt beschreibt

$$\begin{array}{l} \mathit{type}\ \mathit{Pos} = (\mathit{Int}, \mathit{Int}) \\ \mathit{struct}\ \mathit{Point} = \\ \quad \mathit{position} :: \mathit{Pos} \end{array}$$

O'HASKELL führt Subtyp-Polymorphie über Datentypen und Records ein. Wir können also z. B. einen Typ *ColoredPoint* als Subtyp von *Point* definieren.

$$\begin{array}{l} \mathit{data}\ \mathit{Color} = \mathit{Red} \mid \mathit{Green} \mid \mathit{Blue} \\ \mathit{struct}\ \mathit{ColoredPoint} < \mathit{Point} = \\ \quad \mathit{color} :: \mathit{Color} \end{array}$$

Einen Punkt können wir dann wie folgt erzeugen:

$$\begin{array}{l} p = \mathit{struct} \\ \quad \mathit{pos} := (1, 1) \\ \quad \mathit{color} := \mathit{Red} \end{array}$$

Die Auswahl von Record-Elementen erfolgt über die für objektorientierte Sprachen typische Punktnotation. Die Funktion *redpoints* filtert aus einer Liste *pointlist* von Punkten alle roten Punkte heraus:

$$\mathit{redpoints} = (\backslash\, x \rightarrow x\,.\,\mathit{color} = \mathit{Red}) \triangleleft \mathit{pointlist}$$

Ein Selektor kann dabei auch als Funktion in Präfix-Notation verwendet werden:

$$\mathit{redpoints} = (((=)\mathit{Red}) \circ (.\,\mathit{color})) \triangleleft \mathit{pointlist}$$

Objekte werden aus Templates, d. h. Klassen, instanziiert, die den initialen Zustand des Objekts und ein Kommunikationsinterface definieren. Das folgende Codesegment zeigt ein Template *point*.

```
point = template
          pos := (1, 1)
          color := Red
        in struct
          move delta = action
            pos := add (pos, delta)
            where add ((x, y), (dx, dy)) = (x + dx, y + dy)
          newColor newcolor = action
            color := newcolor
          readPosition = request
            return pos
          readColor = request
            return color
```

Die Instanziierung des Templates erzeugt ein Objekt, dessen Zustand durch seine Position und seine Farbe gekennzeichnet ist, und gibt einen Record mit den vier Methoden *move*, *newColor*, *readPosition* und *readColor* als Interface zurück. Eine Methode kann dabei entweder eine asynchrone Aktion sein, sodass der Sender unmittelbar (und damit nebenläufig) fortfährt, oder ein synchroner Request. In diesem Fall wartet der Sender auf einen Antwortwert.

Aktionen, Requests und Templates sind Operationen einer Monade wie *Beh* α, die in O'HASKELL *Cmd* α heißt. Wie wir in Kapitel 17 gesehen haben, berechnen solche Operationen einen extern sichtbaren Wert vom Typ α – *readPosition* gibt beispielsweise die aktuelle Punktposition zurück – und vollziehen weiterhin einen Zustandsübergang auf einem internen Typ. Die Methode *readPosition* ändert in unserem Beispiel den internen Zustand, also die Position, hier nicht, die monadische Operation *move* hingegen setzt die Position neu. Das Template *point* hat den Typ *Cmd Point′*, wobei *Point′* als Recordtyp definiert ist:

```
point :: Cmd Point′
struct Point′ =
  move         :: Pos → Cmd()
  newColor     :: Color → Cmd()
  readPosition :: Cmd Pos
  readColor    :: Cmd Color
```

Wir können nun HASKELLs do-Notation für Berechnungen mit Monaden verwenden und auf diese Weise sequenzielle Folgen von monadischen Operationen darstellen.

```
do p ← point
   p.newColor Green
   p.move (1, 3)
   p.readPosition
```

Wir initialisieren zunächst unser Template *point*, sodass wir nun mit dem Objekt p und vier entsprechenden Methoden arbeiten können. Die Ausführung von $p.newColor\ Green$ setzt die Farbe des Objekts p vom ursprünglichen Wert *Red* nun neu auf *Green*, danach verschieben wir den Punkt um die Distanz $(1, 3)$. Mit *readPosition* geben wir schließlich die aktuelle Position $(2, 4)$ des Punktes als Ergebnis zurück.

22.2.3 OCAML

Genau wie O'HASKELL unterstützt auch die Sprache OCAML [123], die ML erweitert, die meisten objektorientierten Konzepte. Auch hier werden Objekte durch Records implementiert, die Transformation von Objektzuständen nutzt allerdings hier die imperativen Elemente von ML.

22.3 Funktional-logische Programmierung und darüber hinaus

Funktionale, logische und constraint-basierte Programmiersprachen fasst man als *zustandsfreie oder deklarative Sprachen* zusammen. Die Integration verschiedener deklarativer Sprachen ist erfolgreich durchgeführt worden und theoretisch gut begründet.

Ziel bei der Integration funktionaler und logischer Sprachen wie PROLOG war die Kombination ihrer jeweiligen Vorteile: Funktionale Sprachen werten deterministisch aus und sind daher weitaus effizienter als logische Sprachen. Weiterhin möchte man die Eleganz bei der Verwendung von Funktionen höherer Ordnung nutzen. Aber auch logische Sprachen haben spezifische Vorteile und Anwendungsgebiete: Ihre Eleganz und Ausdrucksstärke beruhen auf den Möglichkeiten, mit Funktionsinvertierung und unvollständigen Daten zu arbeiten sowie spezielle Suchstrategien anzuwenden.

In ihrer einfachsten Ausprägung besteht der Hauptunterschied *funktional-logischer Sprachen* zu funktionalen Sprachen darin, dass man in Termen logische, d. h. auch uninstanziierte Variablen als Parameter zulässt. Natürlich muss man den Auswertungsmechanismus entsprechend erweitern. Dabei wird das in funktionalen Sprachen übliche Patternmatching durch die aus der logischen Programmierung bekannte Unifikation der formalen und aktuellen Parameter eines Funktionsaufrufs ersetzt. Der resultierende Mechanismus wird *Narrowing* genannt.

Als Beispiel betrachten wir eine Additionsfunktion auf den natürlichen Zahlen, dargestellt mit Hilfe der Konstruktoren 0 und s (für *succ*):

$add\ 0\ x = x$

$add\ (s\ x)\ y = s\ (add\ x\ y)$

Wir haben die Funktion hier in CURRY-Syntax [67, 69] angegeben. Die Sprache CURRY erweitert HASKELL und wurde entwickelt, um einen Standard im Bereich der funktional-logischen Sprachen zu etablieren.

In HASKELL hätten wir die *add*-Funktion genauso hinschreiben können. Bei der Auswertung eines Ausdrucks dürfen jetzt aber auch uninstanziierte Variablen als Parameter auftreten.

Betrachten wir den Ausdruck $add\ v\ (s\ 0)$, der die Addition der ungebundenen Variablen v und des Wertes $s\ 0$ beschreibt. Durch Unifikation dieses Ausdrucks mit den linken Seiten der Regeln wird eine passende Regel ausgewählt und unser Ausdruck durch die instanziierte rechte Regelseite ersetzt.

$$add\ v\ (s\ 0) \rightsquigarrow_{\{v/(s\ x)\}} s\ (add\ x\ (s\ 0))$$

Wir hätten hier sowohl die erste als auch die zweite Regel wählen können und haben die zweite gewählt. Die berechnete Substitution $\{v\ /\ (s\ x)\}$ haben wir dann auf die rechte Regelseite angewendet. Genauso gehen wir jetzt für den neu berechneten Ausdruck vor, der reduzierte Subterm ist unterstrichen:

$$s\ (\underline{add\ x\ (s\ 0)}) \rightsquigarrow_{\{x/0\}} s\ (s\ 0)$$

Wir berechnen neben einem Ergebnis $s\ (s\ 0)$ auch eine Substitution. In CURRY schreibt man das dann so (id bezeichnet die identische Substitution):

$$\begin{aligned}
&\{id \parallel add\ v\ (s\ 0)\}\\
&\rightsquigarrow \{\{v\ /\ (s\ x)\} \parallel s\ (add\ x\ (s\ 0))\}\\
&\rightsquigarrow \{\{v\ /\ (s\ 0),\ x\ /\ 0\} \parallel s\ (s\ 0)\}
\end{aligned}$$

Wir haben also berechnet, dass $v + 1 = 2$ gilt, wenn v mit 1 instanziiert wird.

Da die Variable v in unserem Ausdruck ungebunden ist, gibt es zu dessen Ableitung hier unendlich viele Möglichkeiten, z. B.

$$\{id \parallel add\ v\ (s\ 0)\} \rightsquigarrow \{\{v\ /\ 0\} \parallel s\ 0\}$$

oder

$$\begin{aligned}
&\{id \parallel add\ v\ (s\ 0)\}\\
&\rightsquigarrow \{\{v\ /\ (s\ x)\} \parallel s\ (add\ x\ (s\ 0))\}\\
&\rightsquigarrow \{\{v\ /\ (s\ (s\ y)),\ x\ /\ (s\ y)\} \parallel s\ (s\ (add\ y\ (s\ 0)))\}\\
&\rightsquigarrow \{\{v\ /\ (s\ (s\ 0)),\ x\ /\ (s\ 0),\ y\ /\ 0\} \parallel s\ (s\ (s\ 0))\}
\end{aligned}$$

usw.

Da CURRY lazy auswertet, ist das aber kein Problem. Die Auswertung erzeugt immer nur so viele Lösungen, wie wir gerade brauchen.

Ein anderer Auswertungsmechanismus ist *Residuation*. Hierbei werden Funktionsaufrufe mit uninstanziierten Variablen als Parameter so lange verzögert, bis alle Variablen von laufenden Prozessen gebunden wurden bzw. bis die Variablen so weit instanziiert sind, dass eine eindeutige Regelauswahl möglich ist.

Werten wir unser Beispiel also mit Residuation aus, dann suspendiert der Aufruf $add\ v\ s(0)$, weil v ungebunden ist. (In rein funktionalen Sprachen bekämen wir eine Fehlermeldung.):

$$\{id \parallel add\ v\ s(0)\} \rightsquigarrow suspend$$

Wir brauchen hier einen weiteren Prozess, der uns eine Belegung von v erzeugt und der mit der Berechnung des Ausdrucks $add\ v\ s(0)$ interagiert. Das kann wieder ein komplexer Funktionsaufruf bzw. Prädikat sein oder ein einfaches Constraint wie im Fall $(v =:= add\ 0\ 0)$. Dabei drückt $=:=$ die Gleichheit

zwischen zwei Ausdrücken aus und der Operator & ist die nebenläufige Konjunktion: Wenn die Auswertung des ersten Ausdrucks suspendiert, wird mit der Auswertung des zweiten begonnen bzw. fortgefahren. Dieser kann den ersten durch eine Bindung gemeinsamer Variabler reaktivieren. In unserem Fall sieht das so aus:

$\{id \parallel add\ v\ (s\ 0) =:= w\ \&\ v =:= \underline{add\ 0\ 0}\}$
$\leadsto \{id \parallel add\ v\ (s\ 0) =:= w\ \&\ v =:= 0\}$

Da die Auswertung der ersten Gleichung suspendiert, wird zunächst die zweite abgeleitet. Den reduzierten Term haben wir wieder unterstrichen. Auch im nächsten Schritt ist eine Reduktion der ersten Gleichung noch nicht möglich, wir fahren deshalb mit der zweiten fort, die die Bindung von v an den Wert 0 erzeugt und diese auf den Gesamtausdruck anwendet. Danach kann der erste Teilausdruck ausgewertet werden:

$\{id \parallel add\ v\ (s\ 0) =:= w\ \&\ \underline{v =:= 0}\}$
$\leadsto \{\{v\ /\ 0\} \parallel add\ 0\ (s\ 0) =:= w\}$
$\leadsto \{\{v\ /\ 0\} \parallel s\ 0 =:= w\}$

Residuation wird z. B. in den Sprachen ESCHER [93], LIFE [13], MOZART/OZ [133] und GOFFIN [33] verwendet. Da hierbei Funktionsaufrufe durch deterministische Reduktionsschritte ausgewertet werden, muss nichtdeterministische Suche durch Prädikate (z. B. in LIFE) oder Disjunktionen (in ESCHER und MOZART/OZ) dargestellt werden. Dieses Auswertungsprinzip ist allerdings unvollständig, d. h., es kann unter Umständen keine Lösungen berechnen, wenn die Argumente von Funktionen nicht ausreichend instanziiert sind, selbst wenn diese nicht zur Lösung beitragen.

Ähnlich wie bei der Reduktion (vgl. Kapitel 1.3) unterscheidet man auch Narrowing-Strategien, wie *innermost*, *outermost* oder *lazy Narrowing*. Für bestimmte Narrowing-Strategien lässt sich unter weiteren einschränkenden Bedingungen Vollständigkeit garantieren [68]. Sprachen mit vollständiger operationaler Semantik, wie BABEL [100] und SLOG [55], basieren auf entsprechenden Narrowing-Strategien. CURRY implementiert beide Auswertungsmechanismen, d. h. Residuation und die lazy Strategie *needed Narrowing*.

Regeln werden ausdrucksstärker, wenn sie außerdem Bedingungen erlauben. Hierbei überträgt man die Idee der Implikation aus der logischen Programmierung in die funktional-logische Welt. Regeln haben dann die Form

$$f\ t_1 \ldots t_n \mid b = e$$

und die Anwendbarkeit einer Regel ist jetzt von der Erfüllbarkeit des Bedingungsteils b abhängig. Man spricht hierbei von bedingtem Narrowing (*conditional Narrowing* [68]). In den Bedingungsteil kann man nun geschickt z. B. Suchvorgänge einbetten.

Haben wir wie üblich eine Funktion *append* zur Verknüpfung von Listen definiert, dann kann man in CURRY die Berechnung des letzten Elements einer Liste wie folgt ausdrücken:

$$last\ l \mid append\ xs\,[x] =:= l = x\ \ where\ x, xs\ free$$

Ist l eine Liste, die wir durch Anhängen eines Elements x an eine Liste xs erhalten können, dann ist x auch tatsächlich das letzte Element der Liste l. Die freien Variablen x und xs muss man dabei in CURRY explizit deklarieren.

Lässt man in den Bedingungen auch Constraints anderer Bereiche zu, so spricht man von *Constraint Functional Logic Programming* [95]. Während die ursprünglichen Bedingungen bzw. Gleichheitsconstraints der Form $e_1 =:= e_2$ in der Sprache selbst ausgewertet werden, werden diese zusätzlichen Constraints nun durch externe Constraint-Löser behandelt.

Das folgende Programm berechnet Pythagoreische Tripel, d. h. Tripel von natürlichen Zahlen, die die Gleichung $a^2 + b^2 = c^2$ erfüllen. Das Argument ist eine Kathete b; das Constraint $a \leq b$ beschränkt die Menge der Lösungen auf eine endliche Anzahl.

$Triple :: Nat \to (Nat \times Nat \times Nat)$

$Triple\ b \mid a \leq_{FD} b, (a * a) + (b * b) =_{FD} (c * c) = (a, b, c)\ where\ a, c\ free$

Die Constraints $a \leq_{FD} b$ und $(a * a) + (b * b) =_{FD} (c * c)$ werden hierbei von einem externen Lösungsmechanismus, einem so genannten Finite-Domain-Constraint-Löser (s. z. B. [97]) behandelt. *Finite-Domain-Constraints* sind dabei im Allgemeinen Gleichungen und Ungleichungen über Variablen, deren Wertebereiche endliche Mengen (hier natürlicher Zahlen) sind. Entsprechende Lösungsmechanismen nutzen diese Eigenschaft aus.

Alternativ zur hier skizzierten engen Integration von Konzepten funktionaler und (constraint-)logischer Sprachen auf der Basis eines einheitlichen Auswertungsmechanismus wird der Ansatz verfolgt, Constraint-Erweiterungen und Suche explizit als Teilsprache der funktionalen Sprache hinzuzufügen. Hier handelt es sich meist um Bibliotheken, wie das SCREAMER Constraint System [130], das COMMON LISP erweitert oder die OCAML Constraint Bibliothek FACILE [26].

22.4 Fazit

Die Integration von verschiedenen Sprachparadigmen in ein gemeinsames Framework ist ein notwendiger und hochaktueller Forschungsgegenstand im Bereich der Programmiersprachen. Die Bedeutung dieser Frage wird noch verstärkt durch die an vielen Stellen erhobene Forderung, applikationsspezifische Modellierungssprachen zu entwickeln.

In ganz pragmatischer Form wird dieses Thema z. B. in dem .NET-Ansatz von Microsoft aufgegriffen. Dieser beschränkt sich aber letztlich nur auf die Kombinierbarkeit von generiertem Code, sodass aus Programmiersicht keinerlei Integration auf der Paradigmenebene erfolgt.

Dass das Thema nach wie vor ein herausfordernder Forschungsgegenstand ist, haben die kurzen Skizzen in den vorangegangenen Abschnitten dieses Kapitels gezeigt. Die Vielfalt der Sprachen und Systeme, die jeweils Teilaspekte des Problemkreises adressieren, ist ein Indiz für die Menge an Arbeit, die hier noch zu leisten ist.

Literatur

1. BIBLIOTHECA OPALICA. `http://uebb.cs.tu-berlin.de/~opal/ocs/doc/html/BibOpalicaManual/BibOpalicaManual.html`. Letzter Zugriff: 17.03.2006.
2. *A GTK+ Binding for Haskell.* `http://www.cse.unsw.edu.au/~chak/haskell/gtk/`. Letzter Zugriff: 17.03.2006.
3. *An Online Bibliography of Scheme-related Research. Continuations and Continuation Passing Style.* `http://library.readscheme.org/page6.html`. Letzter Zugriff: 17.03.2006.
4. *Gtk2Hs – A GUI Library for Haskell based on Gtk.* `http://haskell.org/gtk2hs/`. Letzter Zugriff: 17.03.2006.
5. *Single Assignment C.* `http://www.sac-home.org`. Letzter Zugriff: 17.03.2006.
6. *Specware.* `http://www.specware.org/`. Letzter Zugriff: 17.03.2006.
7. *TclHaskell.* `http://www.dcs.gla.ac.uk/~meurig/TclHaskell/`. Letzter Zugriff: 17.03.2006.
8. *Specware 4.1 Tutorial*, 2004. Kestrel Development Corporation, Kestrel Technology LLC.
9. ACHTEN, PETER, MARKO VAN EEKELEN und RINUS PLASMEIJER: *Generic Graphical User Interfaces.* In: TRINDER, PHILIP W., GREG MICHAELSON und RICARDO PENA (Herausgeber): *15th International Workshop on the Implementation of Functional Languages, IFL 2003*, Band 3145 der Reihe *Lecture Notes in Computer Science*, Seiten 152–167. Springer-Verlag, 2004.
10. ACHTEN, PETER und RINUS PLASMEIJER: *Interactive Functional Objects in Clean.* In: CLACK, CHRIS, KEVIN HAMMOND und ANTONY J.T. DAVIE (Herausgeber): *Proceedings of 9th International Workshop on Implementation of Functional Languages – IFL'97*, Band 1467 der Reihe *Lecture Notes in Computer Science*, Seiten 304–321. Springer-Verlag, 1998.
11. ACHTEN, PETER und MARTIN WIERICH: *A Tutorial to the Clean Object I/O Library – Version 1.2.* Technischer Bericht, University of Nijmegen, February 2000.
12. ADAMS, DOUGLAS: *Das Restaurant am Ende des Universums.* Ullstein, 1985.
13. AÏT-KACI, HASSAN: *An Overview of LIFE.* In: SCHMIDT, JOACHIM W. und ANATOLY A. STOGNY (Herausgeber): *Proc. Workshop on Next Generation Information System Technology*, Band 504 der Reihe *Lecture Notes in Computer Science*, Seiten 42–58. Springer-Verlag, 1990.

14. Armstrong, Joe, Robert Virding, Claes Wikstrom und Mike Williams: *Concurrent Programming in Erlang.* Prentice Hall, Zweite Auflage, 1996.
15. Augustsson, Lennart: *Cayenne - a Language with Dependent Types.* In: *Third ACM SIGPLAN International Conference on Functional Programming, ICFP*, Band 34 (1) der Reihe *SIGPLAN Notices*, Seiten 239–250. ACM, 1999.
16. Backhouse, Roland, Patrik Jansson, Johan Jeuring und Lambert Meertens: *Generic Programming — An Introduction.* In: *Advanced Functional Programming*, Band 1608 der Reihe *Lecture Notes in Computer Science*, Seiten 28–115. Springer-Verlag, 1999.
17. Barendregt, Henk: *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.
18. Bauer, F. L. und H. Wössner: *Algorithmische Sprache und Programmentwicklung.* Springer-Verlag, 1981.
19. Bidoit, Michel und Peter D. Mosses: *CASL User Manual*, Band 2900 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag, 2004.
20. Bird, Richard: *Introduction to Functional Programming using Haskell.* Prentice Hall, Zweite Auflage, 1998.
21. Bird, Richard und Oege de Moor: *Algebra of Programming.* Prentice Hall, 1997.
22. Bird, Richard und Philip Wadler: *Introduction to Functional Programming.* Prentice Hall, 1988.
23. Birkhoff, Garrett: *Lattice Theory.* American Mathematical Society, Dritte Auflage, 1967.
24. Bourbaki, Nicolas: *Éléments de mathématique.* Hermann, Paris.
25. Breitinger, Silvia, Rita Loogen, Yolanda Ortega-Mallén und Ricardo Peña: *Eden - Language Definition and Operational Semantics.* Reihe Informatik TR-96-10, Philipps Universität Marburg, Fachbereich Mathematik und Informatik, 1998.
26. Brisset, Pascal und Nicolas Barnier: *FaCiLe: a Functional Constraint Library.* In: *Proceedings of the MultiCPL'02 Workshop on Multiparadigm Constraint Programming Languages*, Seiten 7–22, September 2002.
27. Burke, Edmund K. und Graham Kendall (Herausgeber): *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques.* Springer-Verlag, 2005.
28. Cai, Jiazhen und Robert Paige: *Program Derivation by Fixed Point Computation.* Science of Computer Programming, 11(3):197–261, April 1989.
29. Cai, Jiazhen und Robert Paige: *Towards Increased Productivity of Algorithm Implementation.* In: Notkin, David (Herausgeber): *Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering*, Band 18 (5) der Reihe *ACM SIGSOFT Software Engineering Notes*, Seiten 71–78, December 1993.
30. Cardelli, Luca und Peter Wegner: *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, 17(4):471–522, 1985.
31. Carlsson, Magnus und Thomas Hallgren: *Fudgets – Purely Functional Processes with Applications to Graphical User Interfaces.* Doktorarbeit, Chalmers University of Technology, Göteborg University, 1998.
32. Chakravarty, Manuel M. T. und Gabriele C. Keller: *Einführung in die Programmierung mit Haskell.* Pearson, 2004.

33. Chakravarty, Manuel M.T., Yike Guo, Martin Köhler und Hendrik C. R. Lock: Goffin*: Higher-Order Functions Meet Concurrent Constraints.* Science of Computer Programming, 30(1-2):157–199, 1998.
34. Church, Alonzo: *The Calculi of Lambda Conversion.* Princeton University Press, 1941.
35. Clavel, Manuel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer und José F. Quesada: *Maude: Specification and Programming in Rewriting Logic.* Theoretical Computer Science, 285(2):187–243, 2002.
36. Cole, Murray: *Algorithmic Skeletons: Structured Management of Parallel Computation.* The MIT Press, 1989.
37. Cormen, Thomas H., Charles E. Leiserson und Ronald L. Rivest: *Introduction to Algorithms.* The MIT Press, 2001.
38. Costa, Vítor Santos, David H.D. Warren und Rang Yang: *Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism.* In: *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Band 26(7) der Reihe *SIGPLAN Notices*, Seiten 83–93. ACM Press, 1991.
39. Davey, B.A. und H.A. Priestley: *Introduction to Lattices and Order.* Cambridge University Press, Zweite Auflage, 2002.
40. Dean, Jeffrey und Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters.* In: *6th Symposium on Operating System Design and Implementation, OSDI*, Seiten 137–150, 2004.
41. Demers, Francois-Nicola und Jacques Malenfant: *Reflection in Logic, Functional and Object-oriented Programming: A Short Comparative Study.* In: *IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, Seiten 29–38, 1995.
42. Didrich, Klaus, Wolfgang Grieskamp, Florian Schintke, Till Tantau und Baltasar Trancòn y Widemann: *Reflections in Opal.* In: *11th International Workshop on Implementation of Functional Languages, IFL*, Band 1868 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag, 2000.
43. Ehrig, Hartmut und Bernd Mahr: *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.
44. Ehrig, Hartmut und Bernd Mahr: *Fundamentals of Algebraic Specification 2.* Springer-Verlag, 1990.
45. Ehrig, Hartmut, Bernd Mahr, Felix Cornelius, Martin Grosse-Rhode und Philip Zeitz: *Mathematisch-strukturelle Grundlagen der Informatik.* Springer-Verlag, 2001.
46. Elliott, Conal und Paul Hudak: *Functional Reactive Animation.* ACM SIGPLAN Notices, 32(8):263–273, 1997. Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming – ICFP.
47. Erwig, Martin: *Grundlagen funktionaler Programmierung.* Oldenbourg, 1999.
48. Fiadeiro, José L. (Herausgeber): *Categories for Software Engineering.* Springer-Verlag, 2005.
49. Fiadeiro, José L., Neil Harman, Markus Roggenbach und Jan J.M.M. Rutten (Herausgeber): *Algebra and Coalgebra in Computer Science*, Band 3629 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag, 2005.
50. Field, Anthony J. und Peter G. Harrison: *Functional Programming.* Addison-Wesley, 1988.

51. Finne, Sigbjorn und Simon Peyton Jones: *Composing the User Interface with Haggis.* In: Launchbury, John, Erik Meijer und Tim Sheard (Herausgeber): *Advanced Functional Programming*, Band 1129 der Reihe *Lecture Notes in Computer Science*, Seiten 1–37. Springer-Verlag, 1996.
52. Flanagan, David (Herausgeber): *Java Foundation Classes in a Nutshell.* O'Reilly, 2000.
53. Frauenstein, Thomas, Wolfgang Grieskamp, Peter Pepper und Mario Südholt: *Communicating Functional Agents and their Application to Graphical User Interfaces.* Technischer Bericht TR 95-19, Technische Universität Berlin, 1996.
54. Frauenstein, Thomas, Wolfgang Grieskamp, Peter Pepper und Mario Südholt: *Communicating Functional Agents and their Application to Graphical User Interfaces.* In: Bjørner, Dines, Manfred Broy und Igor V. Pottosin (Herausgeber): *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference*, Band 1181 der Reihe *Lecture Notes in Computer Science*, Seiten 386–397. Springer-Verlag, 1996.
55. Fribourg, Laurent: *SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting.* In: *Proceedings of the 1985 IEEE Symposium on Logic Programming*, Seiten 172–184, 1985.
56. Gentleman, W. Morven: *Some Complexity Results for Matrix Computations on Parallel Processors.* Journal of the ACM, 25(1):112–115, 1978.
57. Girard, Jean-Yves: *Une extension de l'intérpretation de Godel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types.* In: *Second Scandinavian Logic Symposium*, Band 63 der Reihe *Studies in Logic and the Foundations of Mathematics*, Seiten 63–92. North-Holland, 1971.
58. Girard, Jean-Yves: *Linear Logic.* Theoretical Computer Science, 50:1–102, 1987.
59. Girard, Jean-Yves: *Light Linear Logic.* Information and Computation, 143(2):175–204, 1998.
60. Gordon, Michael J. C.: *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.
61. Grabmüller, Martin: *Multiparadigmen-Programmiersprachen.* Technischer Bericht 2003-15, Technische Universität Berlin, October 2003.
62. Gries, David: *The Science of Programming.* Springer-Verlag, 1981.
63. Gunter, Carl A.: *Semantics of Programming Languages: Structures and Techniques.* The MIT Press, 1992.
64. Gurevich, Yuri: *Evolving Algebras: An Attempt to Discover Semantics.* In: Rozenberg, Grzegorz und Arto Salomaa (Herausgeber): *Current Trends in Theoretical Computer Science*, Seiten 266–292. World Scientific, 1993.
65. Gurevich, Yuri: *Evolving Algebras 1993: Lipari Guide.* In: Börger, E. (Herausgeber): *Specification and Validation Methods*, Seiten 9–36. Oxford University Press, 1995.
66. Hammond, Kevin und Greg Michaelson (Herausgeber): *Research Directions in Parallel Functional Programming.* Springer-Verlag, 1999.
67. Hanus, M., S. Antoy, H. Kuchen, F.J. López-Fraguas, W. Lux, J.J. Moreno-Navarro und F. Steiner: *Curry. An Integrated Functional Logic Language.* Technischer Bericht, Version 0.8 of April 15, 2003.
68. Hanus, Michael: *The Integration of Functions into Logic Programming: From Theory to Practice.* Journal of Logic Programming, 19&20:583–628, 1994.

69. Hanus, Michael: *A Unified Computation Model for Functional and Logic Programming*. In: *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 80–93. ACM, 1997.
70. Hawking, Stephen W.: *Eine kurze Geschichte der Zeit*. Rowohlt, 1998.
71. Hawking, Stephen W. und Leonard Mlodinow: *Die kürzeste Geschichte der Zeit*. Rowohlt, 2005.
72. Heering, Jan, P. R. H. Hendriks, Paul Klint und J. Rekers: *The Syntax Definition Formalism SDF – Reference Manual*. SIGPLAN Notices, 24(11):43–75, 1989.
73. Hermann, Martin: *Numerische Mathematik*. Oldenbourg, 2001.
74. Herrtwich, R.G. und G. Hommel: *Nebenläufige Programme*. Springer-Verlag, 1998.
75. Hill, Jonathan M. D., Keith M. Clarke und Richard Bornat: *The vectorization monad*. In: *First International Symposium on Parallel Symbolic Computation – PASCO*, Seiten 204–214. World Scientific Publishing Company, Hagenberg/Linz, Austria, 1994.
76. Hill, Steve: *Combinators for Parsing Expressions*. Journal of Functional Programming, 6(3):445–463, May 1996.
77. Hinze, Ralf und Ross Paterson: *Finger trees: A Simple General-purpose Data Structure*. Journal of Functional Programming, 16(2):197–217, 2006.
78. Hoeve, Willem Jan van: *The alldifferent Constraint: A Survey*. In: *Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.
79. Hood, Robert und Robert Melville: *Real-Time Queue Operation in Pure LISP*. Information Processing Letters, 13(2):50–54, November 1981.
80. Hopcroft, John E., Jeffrey D. Ullman und Rajeev Motwani: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, Zweite Auflage, 2002.
81. Hudak, Paul: *The Haskell School of Expression*. Cambridge University Press, 2000.
82. Huggins, James K. und Charles Wallace: *An Abstract State Machine Primer*. Technischer Bericht CS-TR-02-04, Computer Science Department, Michigan Technological University, 2002.
83. Hughes, John: *Generalising Monads to Arrows*. Sience of Computer Programming, 37(1-3):67–111, 2000.
84. Hughes, John und Jan Sparud: *Haskell++: An Object-Oriented Extension of Haskell*. In: *Proceedings of Haskell Workshop, La Jolla, California*, YALE Research Report DCS/RR-1075, 1995.
85. Hutton, Graham: *Higher-Order Functions for Parsing*. Journal of Functional Programming, 2(3):323–343, July 1992.
86. Hutton, Graham und Erik Meijer: *Monadic Parsing in Haskell*. Journal of Functional Programming, 8(4):437–444, July 1998.
87. Jeuring, Johan und Patrik Jansson: *Polytypic Programming*. In: Launchbury, John, Erik Meijer und Tim Sheard (Herausgeber): *Advanced Functional Programming*, Band 1129 der Reihe *Lecture Notes in Computer Science*, Seiten 68–114. Springer-Verlag, 1996.
88. Koopman, P.W.M. und R. Plasmeijer: *Efficient Combinator Parsers*. In: Hammond, K., A.J.T. Davie und C. Clack (Herausgeber): *10th International Workshop on Implementation of Functional Languages (IFL'98)*, Band 1595 der Reihe *Lecture Notes in Computer Science*, Seiten 120–136. Springer-Verlag, 1999.

89. Kuchen, Herbert: *Datenparallele Programmierung von MIMD-Rechnern mit verteiltem Speicher*. RWTH Aachen, 1995. Habilitationsschrift.
90. Landin, Peter J.: *A Correspondence between ALGOL 60 and Church's Lambda Notation*. Communications of the ACM, 8(2):89–101, 1965.
91. Lane, Saunders Mac: *Categories for the Working Mathematician*. Springer-Verlag, Zweite Auflage, 1998.
92. Leijen, Daan: *wxHaskell – A portable and concise GUI library for Haskell*. In: *Proceedings of the ACM SIGPLAN workshop on Haskell*, Seiten 57–68. ACM Press, 2004.
93. Lloyd, J.W.: *Programming in an Integrated Functional and Logic Language*. Journal of Functional and Logic Programming, 1999(3), March 1999.
94. Loidl, Hans-Wolfgang, Fernando Rubio, Norman Scaife, Kevin Hammond, Susumu Horiguchi, Ulrike Klusik, Rita Loogen, Greg Michaelson, Ricardo Peña, Steffen Priebe, Álvaro J. Rebón Portillo und Philip W. Trinder: *Comparing Parallel Functional Languages: Programming and Performance*. Higher-Order and Symbolic Computation, 16(3):203–251, September 2003.
95. López-Fraguas, F.J.: *A General Scheme for Constraint Functional Logic Programming*. In: Kirchner, H. und G. Levi (Herausgeber): *Algebraic and Logic Programming – ALP'92*, Band 632 der Reihe *Lecture Notes in Computer Science*, Seiten 213–227. Springer-Verlag, 1992.
96. Manna, Zohar: *Mathematical Theory of Computation*. McGraw-Hill, 1974.
97. Marriott, Kim und Peter J. Stuckey: *Programming with Constraints. An Introduction*. The MIT Press, 1998.
98. Missura, Stephan Albert: *Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing*. Doktorarbeit, Eidgenössische Technische Hochschule Zürich, 1997.
99. Moggi, E.: *Computational lambda-calculus and monads*. In: *IEEE Symposium on Logic in Computer Science*, 1989.
100. Moreno-Navarro, J.J. und M. Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*. Journal of Logic Programming, 12:191–223, 1992.
101. Mosses, Peter D. (Herausgeber): *CASL Reference Manual*, Band 2960 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
102. Myers, Colin, Chris Clack und Ellen Poon: *Programming with Standard ML*. Prentice Hall, 1993.
103. Nipkow, Tobias, Lawrence C. Paulson und Markus Wenzel: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Band 2283 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
104. Nitsche, Thomas: *Skeleton Implementations based on Generic Data Distributions*. In: Gorlatch, Sergei und Christian Lengauer (Herausgeber): *Workshop on Constructive Methods for Parallel Programming – CMPP*, Band 10 der Reihe *Advances in Computation: Theory and Practice*. Nova Science, 2002.
105. Nitsche, Thomas: *Data Distribution and Communication Management for Parallel Systems*. Doktorarbeit, Technische Universität Berlin, 2005.
106. Nordlander, Johan: *Polymorphic Subtyping in O'Haskell*. Science of Computer Programming, 43(2-3):93–127, 2002.
107. Okasaki, Chris: *Purely Functional Data Structures*. Cambridge University Press, 1998.

108. Partridge, Andrew und David Wright: *Predictive Parser Combinators Need four Values to Report Errors.* Journal of Functional Programming, 6(2):355–364, March 1996.
109. Partsch, Helmuth: *Specification and Transformation of Programs.* Springer-Verlag, 1990.
110. Paulson, Lawrence C.: *Isabelle — A Generic Theorem Prover*, Band 828 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag, 1994.
111. Pepper, Peter: *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER.* Springer-Verlag, Zweite Auflage, 2003.
112. Pepper, Peter: *How To Obtain Powerful Parsers That Are Elegant* and *Practical.* Technischer Bericht 2004/01, Technische Universität Berlin, 2004.
113. Pepper, Peter und Douglas R. Smith: *A High-level Derivation of Global Search Algorithms (with Constraint Propagation).* Science of Computer Programming, 28(2-3):247–271, 1997.
114. Pepper, Peter und Mario Südholt: *Deriving Parallel Numerical Algorithms using Data Distribution Algebras: Wang's Algorithm.* In: *Proceedings of the 30th Annual Hawaii International Conference on System Sciences (HICSS).* IEEE Computer Society, 1997.
115. Pepper, Peter und Mario Südholt: *Deriving Parallel Numerical Algorithms using Data Distribution Algebras: Wang's Algorithm.* Technischer Bericht TR 96-2, Technische Universität Berlin, 1996.
116. Peyton Jones, Simon, Andrew Gordon und Sigbjorn Finne: *Concurrent Haskell.* In: *23rd ACM Symposium on Principles of Programming Languages*, Seiten 295–308, 1996.
117. Pierce, Benjamin C.: *Types and Programming Languages.* The MIT Press, 2002.
118. Pierce, Benjamin C. (Herausgeber): *Advanced Topics in Types and Programming Languages.* The MIT Press, 2005.
119. Plasmeijer, Rinus und Marko van Eekelen: *Clean Version 2.0 Language Report.* Department of Software Technology, University of Nijmegen, 2001.
120. Pointon, Robert F., Philip W. Trinder und Hans-Wolfgang Loidl: *The Design and Implementation of Glasgow Distributed Haskell.* In: *12th International Workshop on Implementation of Functional Languages (IFL)*, Band 2011 der Reihe *Lecture Notes in Computer Science*, Seiten 53–70. Springer-Verlag, 2000.
121. Quarteroni, A., R. Sacco und F. Saleri: *Numerische Mathematik (Bd. 1+2).* Springer-Verlag, 2002.
122. Rabhi, Fethi A. und Guy Lapalme: *Algorithms: A Functional Programming Approach.* Addison-Wesley, 1999.
123. Rémy, Didier und Jerome Vouillon: *Objective ML: An Effective Object-Oriented Extension to ML.* TAPOS - Theory and Practice of Objects Systems, 4(1):27–50, 1998.
124. Reynolds, John C.: *Towards a Theory of Type Structure.* In: Robinet, Bernard (Herausgeber): *Programming Symposium, Proceedings Colloque Sur La Programmation*, Band 19 der Reihe *Lecture Notes in Computer Science*, Seiten 408–423. Springer-Verlag, 1974.
125. Sage, Meurig: *FranTk – A declarative GUI Language for Haskell.* ACM SIGPLAN Notices, 35(9):106–117, 2000. Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP.

126. Schmidt, David A.: *Denotational Semantics – A Methodology for Language Development.* William C. Brown Publishers, 1988.
127. Scholz, Sven-Bodo: *Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting.* Journal of Functional Programming, 13(6):1005–1059, 2003.
128. Schulte, Wolfram: *Effiziente und korrekte Übersetzung strikter applikativer Programmiersprachen.* Doktorarbeit, Technische Universität Berlin, 1992.
129. Sedgewick, Robert: *Algorithmen.* Pearson Studium, 2002.
130. Siskind, J.M. und D.A. McAllester: *Nondeterministic LISP as a Substrate for Constraint Logic Programming.* In: *Proceedings of 11th National Conference on Artificial Intelligence*, Seiten 133–138. The AAAI Press/The MIT Press, 1993.
131. Smith, Brian C.: *Reflection and Semantics in a Procedural Language.* Technischer Bericht MIT-LCS-TR-272, MIT Laboratory for Computer Science, 1982.
132. Smith, Brian C.: *Reflection and Semantics in Lisp.* In: *14th Annual ACM Symposium on Principles of Programming Languages, POPL*, Seiten 23–35, 1984.
133. Smolka, G.: *The Oz Programming Model.* In: Leeuwen, J. van (Herausgeber): *Computer Science Today*, Band 1000 der Reihe *Lecture Notes in Computer Science*, Seiten 324–343. Springer-Verlag, 1995.
134. Stoer, Josef: *Numerische Mathematik.* Springer-Verlag, 2005.
135. Stoy, Joseph E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.
136. Swierstra, D. und P.R. Azero Alcocer: *Fast, Error-Correcting Parser Combinators: a Short Tutorial.* In: Pavelka, J., G. Tel und M. Bartosek (Herausgeber): *SOFSEM'99: Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics*, Band 1725 der Reihe *Lecture Notes in Computer Science*, Seiten 112–131. Springer-Verlag, 1999.
137. Südholt, Mario: *The Transformational Derivation of Parallel Programs using Data-Distribution Algebras and Skeletons.* Doktorarbeit, Technische Universität Berlin, 1997.
138. Thiemann, Peter: *Grundlagen der Funktionalen Programmierung.* Teubner Verlag, 1997.
139. Thompson, Simon: *Type Theory and Functional Programming.* Addison-Wesley, 1991.
140. Trinder, Philip W., Kevin Hammond, Hans-Wolfgang Loidl und Simon L. Peyton Jones: *Algorithms + Strategy = Parallelism.* Journal of Functional Programming, 8(1):23–60, 1998.
141. Trinder, Philip W., Hans-Wolfgang Loidl und Robert F. Pointon: *Parallel and Distributed Haskells.* Journal of Functional Programming, 12(4&5):469–510, July 2002.
142. Wadler, Philip: *Comprehending monads.* Mathematical Structures in Computer Science, 2:461–493, 1992.
143. Wand, Mitchell: *Continuation-Based Program Transformation Strategies.* Journal of the ACM, 27(1):164–180, 1980.
144. Warren, David H.D.: *The Andorra Principle.* Presented at the Gigalips Workshop, Swedish Institute of Computer Science (SICS), Stockholm, Sweden, 1988.

145. Wieland, Jacob: *Parsing Mixfix Expressions – How to Deal with User-Defined Mixfix Operators Efficiently.* Doktorarbeit, Technische Universität Berlin, to appear.
146. Winskel, Glynn: *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, 1993.

Index