Skriptum zur Vorlesung

Strukturen von Programmiersprachen

Martin Odersky

Universität Karlsruhe

12. April 1995

Teil I

Beschreibungstechniken

Kapitel 1

Kontextfreie Syntax

1.1 Konkrete Syntax

Kontextfreie Syntax wird in der Regel durch kontextfreie Grammatiken beschrieben.

Beispiel 1.1 Eine Grammatik für arithmetische Ausdrücke.

$$\begin{array}{lcl} E & ::= & E \,\texttt{+}\, T \mid E \,\texttt{-}\, T \mid T \\ T & ::= & T \,\texttt{*}\, F \mid T \,\texttt{/}\, F \mid F \\ F & ::= & N \mid \texttt{(}E\texttt{)} \\ N & ::= & \texttt{int} \mid \texttt{real} \end{array}$$

Oft wird die Beschreibung kontextfreier Syntax aufgeteilt in zwei Grammatiken. Eine Grammatik beschreibt dabei die *Makrosyntax*, mit Worten als Terminalsymbolen. Eine weitere Grammatik beschreibt die *Mikrosyntax*, die den Aufbau der einzelnen Terminalsymbole der Makrosyntax festlegt. In der Regel sind Leerzeichen nur für die Mikrosyntax signifikant. Ebenfalls nur für die Mikrosyntax gilt die Regel, daß jeweils die längstmögliche Zeichenkette zu einem Wort zusammengefaßt wird ("longest match rule").

Beispiel 1.2 Die Grammatik für arithmetische Ausdrücke wird um folgende Mikrosyntax ergänzt.

$$\begin{array}{lcl} \texttt{int} & ::= & \texttt{digit} \mid \texttt{int digit} \\ \texttt{real} & ::= & \texttt{int . int} \\ \texttt{digit} & ::= & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \end{array}$$

Bemerkung. Terminale und Nichtterminale der Makrosyntax werden durch ihren Zeichensatz unterschieden: mathematische Zeichen für Nichtterminale, z. B. E, F, Schreibmaschinenschrift für Terminale, zb `int`, `+`. Für die Mikrosyntax folgen wir der Konvention, daß Ein-Zeichen-Symbole Terminale sind und Mehr-Zeichen-Symbole Nichtterminale darstellen.

Abkürzungen

$$\begin{array}{lcl} \{X\} & : & \text{Endlich viele Vorkommnisse von } X \ (0,\ 1,\ 2,\ \ldots) \\ [X] & : & \text{Null oder ein Vorkommnis von } X \\ X \mid Y & : & X \text{ oder } Y \end{array}$$

Beispiel 1.3 Mit diesen Abkürzungen können arithmetische Ausdrücke nun so definiert werden:

$$\begin{array}{lcl} E & ::= & T\ \{\texttt{+}\ T \mid \texttt{-}\ T\} \\ T & ::= & F\ \{\texttt{*}\ F \mid \texttt{/}\ F\} \\ F & ::= & N \mid (E) \\ N & ::= & \texttt{int} \mid \texttt{real} \end{array}$$

Diese Schreibweise hat den Vorteil, daß Linksrekursion vermieden wird.

1.2 Abstrakte Syntax

Kontextfreie Syntax wird oft unterschieden in *konkrete Syntax* und *abstrakte Syntax*. Die konkrete Syntax dient als Grundlage der Zerteilung und sollte deshalb *eindeutig* sein.

Definition 1.4 Eine Grammatik G ist eindeutig, wenn jede Zeichenkette höchstens eine Ableitung in G hat.

Neben der Eindeutigkeit wird oft noch die Existenz eines effizienten Zerteilers verlangt. Im Prinzip kann allerdings jede kontextfreie Sprache zerteilt werden.

Satz 1.5 (Earley) Jede kontextfreie Sprache hat einen Zerteiler der Komplexität $O(N^3)$, wo N die Länge der Eingabe ist.

Abstrakte Syntax muß nicht eindeutig sein, da vom Strukturbaum als Eingabe ausgegangen wird. Dafür sollte sie so klar und kompakt wie möglich sein.

Beispiel 1.6 Abstrakte Syntax arithmetischer Ausdrücke.

$$\begin{array}{lcl} e & ::= & e + e \mid e - e \mid e * e \mid e/e \mid n \\ n & ::= & i \mid r \\ i & ::= & i\,d \mid d \\ r & ::= & i.i \\ d & ::= & 0 \mid 1 \mid \ldots \mid 9 \end{array}$$

Wir unterscheiden zwischen konkreter und abstrakter Syntax dadurch, daß Nichtterminale der abstrakten Syntax kleingeschrieben werden.

Es ist möglich, direkt in abstrakter Syntax zu programmieren, z. B. durch Verwendung eines Struktureditors. Häufiger allerdings führt der Compiler eine Übersetzung von konkreter zu abstrakter Syntax durch. Diese Phase wird syntaktische Analyse genannt. Im weiteren werden Programmtexte nur noch in abstrakter Syntax dargestellt.

Kapitel 2

Kontextabhängige Syntax

Die meisten Sprachen sind durch eine kontextfreie Syntax nicht zu beschreiben (arithmetische Ausdrücke bilden hier eine Ausnahme). Eine typische Formulierung, die nicht in einer kontextfreien Grammatik dargestellt werden kann, ist z. B. die, daß jeder Bezeichner vereinbart sein muß. Es ist im Prinzip möglich, solche Eigenschaften mit einer kontextabhängigen Grammatik zu beschreiben. Z. B. wurden Van-Wijngaarden-Grammatiken für die Beschreibung der Sprache Algol 68 eingesetzt [19]. Allerdings hat sich herausgestellt, daß andere Beschreibungsmittel günstiger sind, d. H. zu kompakteren und besser lesbaren Beschreibungen führen. Wir werden kontextabhängige Syntax durch *Deduktionssysteme* beschreiben.

2.1 Deduktionssysteme

Definition 2.1 Ein Deduktionssystem besteht aus

1. einer Menge von *Klauseln* K. Beispiel:

   $$\{e : \tau \mid e \in Exp, \tau \in Type\},$$

   wo $Type = \{\mathsf{int}, \mathsf{real}\}$ und Exp die Menge aller (abstrakten) arithmetischen Ausdrücke ist, wie sie im letzten Kapitel definiert wurde;

2. einer Menge von *Axiomen* A, die Untermenge der Menge aller Klauseln ist. Beispiel:

   $$\{0 : \mathsf{int}, 1 : \mathsf{int}, 2 : \mathsf{int}, \ldots\}$$

3. einer Menge von *Inferenzregeln*

$$\frac{K_1 \ldots K_n}{L},$$

wo die *Prämissen* K_1, ..., K_n und die *Schlußfolgerung* L Klauseln sind.

Axiome und Regeln werden oft auch als *Schemata* geschrieben. So ist das Schema $i : \mathsf{int}$ äquivalent zu $\{0 : \mathsf{int}, 1 : \mathsf{int}, 2 : \mathsf{int}, \ldots\}$, und die Regelmenge

$$\frac{e_1 : \tau \quad e_2 : \tau}{e_1 + e_2 : \tau}$$

umfaßt z. B.

$$\frac{1+2 : \mathsf{int} \quad 3 : \mathsf{int}}{(1+2)+3 : \mathsf{int}} \quad \text{und} \quad \frac{1.0 : \mathsf{real} \quad 1.0 : \mathsf{real}}{1.0 + 1.0 : \mathsf{real}} .$$

Definition 2.2 Eine Klausel K is *gültig* (oder *ableitbar*) in einem Deduktionssystem, wenn es einen (endlichen) Baum mit Klauseln als Knoten gibt, so daß

1. K die Wurzel des Baumes ist,
2. jedes Blatt des Baumes ein Axiom ist,
3. für alle internen Knoten L mit Nachfolgern K_1,...,K_n gilt:

$$\frac{K_1 \ \ldots \ K_n}{L}$$

ist eine Inferenzregel.

Anmerkung (1): Oft sind Ableitungen auch Schemata.

Anmerkung (2): Manchmal haben Klauseln Seitenbedingungen B, die nicht der Syntax von Klauseln genügen müssen:

$$\frac{K_1 \ \ldots \ K_n}{L} (B) \ .$$

Solche Seitenbedungungen heißen *außerlogisch*, im Gegensatz zu den Klauseln, die auch *logische Aussagen* heißen. Die Regel für Gültigkeit einer Klausel wird dann dahingehend modifiziert, daß die Seitenbedingungen aller Inferenzregeln einer Ableitung erfüllt sein müssen.

Anmerkung (3): Dem besseren Verständnis wegen werden Axiome und Inferenzregeln oft mit Namen versehen, die in Klammern vor oder nach das Axiom oder die Regel geschrieben werden.

Beispiel 2.3 Man betrachte ein Deduktionssystem für arithmetische Ausdrücke mit Axiomen und Regeln

$$(i) \quad i : \mathsf{int}$$

$$(+) \quad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}} .$$

Dann ist $(1+2)+3 : \mathsf{int}$ gültig, mit Ableitung

$$\frac{\dfrac{1 : \mathsf{int} \quad 2 : \mathsf{int}}{1 : 2 : \mathsf{int}}(+) \qquad 3 : \mathsf{int}}{(1+2)+3 : \mathsf{int}}(+) .$$

Deduktionssysteme kommen aus der Beweistheorie, einem Zweig der mathematischen Logik. Ein Beispiel, das für uns später noch wichtig wird, ist ein Deduktionsystem, das positive Aussagenlogik beschreibt.

Die abstrakte Syntax positiver Aussagen ist

$$p ::= a \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid \mathit{true} \ .$$

Hier ist *true* ein Symbol und a steht für eine (abzählbar unendliche) Menge von *Aussagenvariablen*.

Die Klauseln des Deduktionssystems haben die Form

$$\Gamma \vdash p \ .$$

Hier ist p eine positive Aussage, und Γ ist eine endliche Menge $\{p_1, ..., p_n\}$ von positiven Aussagen p_1, ... , p_n. Informell ist $\Gamma \vdash p$ gültig, wenn p unter den Annahmen Γ gefolgert werden kann. Das ($\vdash$)-Symbol entspricht also der Implikation $\Rightarrow$. Allerdings ist $\Rightarrow$ *logisch*, d. H. Teil der Klauselsprache, während $\vdash$ *meta-logisch* ist.

$$
\begin{array}{ll}
(\text{TAUT}) & \Gamma \cup \{p\} \vdash p \\[1ex]
(\text{TRUE}) & \Gamma \vdash \mathit{true} \\[1ex]
(\wedge\text{-I}) & \dfrac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q} \\[2ex]
(\wedge\text{-E}) & \dfrac{\Gamma \vdash p \wedge q}{\Gamma \vdash p} \qquad \dfrac{\Gamma \vdash p \wedge q}{\Gamma \vdash q} \\[2ex]
(\vee\text{-I}) & \dfrac{\Gamma \vdash p}{\Gamma \vdash p \vee q} \qquad \dfrac{\Gamma \vdash q}{\Gamma \vdash p \vee q} \\[2ex]
(\vee\text{-E}) & \dfrac{\Gamma \vdash p \vee q \quad \Gamma \cup \{p\} \vdash r \quad \Gamma \cup \{q\} \vdash r}{\Gamma \vdash r} \\[2ex]
(\Rightarrow\text{-I}) & \dfrac{\Gamma \cup \{p\} \vdash q}{\Gamma \vdash p \Rightarrow q} \\[2ex]
(\Rightarrow\text{-E}) & \dfrac{\Gamma \vdash p \Rightarrow q \quad \Gamma \vdash p}{\Gamma \vdash q}
\end{array}
$$

Abbildung 2.1: Positive Aussagenlogik

Die Gültigkeit positiver Aussagen wird nun durch die Axiome und Regeln von Abbildung 2.1 festgelegt.

Anmerkung: Für jedes Konstrukt gibt es mindestens eine Einführungsregel und mindestens eine Eliminationsregel. Z. B. hat ($\Rightarrow$-I) eine Implikation in der Schlußfolgerung und ist damit Einführungsregel für $\Rightarrow$. ($\Rightarrow$-E) hat eine Implikation in einer Prämisse und ist damit Eliminationsregel für $\Rightarrow$.

2.2 Erweiterte arithmetische Ausdrücke

Wir erweitern arithmetische Ausdrücke um Variablen. Eine Variable x steht für eine Unterberechnung e, die in einem Ausdruck der Form $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ definiert wird. Wir verlangen, daß jede benutzte Variable deklariert wird, und legen die üblichen block-strukturierten Sichtbarkeitsregeln zugrunde.

Die kontextabhängige Syntax dieser Sprache wird durch ein Regelsystem gegeben, dessen Klauseln die Form

$$\Gamma \vdash e : \tau$$

haben. Hier ist e ein Ausdruck, der durch folgende abstrakte Syntax bestimmt wird:

$$e ::= n \mid e + e \mid e - e \mid e * e \mid e/e \mid x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e\ .$$

Weiter ist der Typ τ ein Element von $\{\mathsf{int}, \mathsf{real}\}$. Schließlich ist die *Typumgebung* (oder *Typothese*) Γ eine Folge von Typbindungen $x : \tau$. Syntaktisch kann das folgendermaßen ausgedrückt werden:

$$\Gamma \ ::= \ \{\} \mid x : \tau \mid \Gamma, \Gamma$$

mit den Gesetzen, daß $(,)$ assoziativ mit $\{\}$ als Identität ist:

$$
\begin{array}{rcl}
(\Gamma_1, \Gamma_2), \Gamma_3 & = & \Gamma_1, (\Gamma_2, \Gamma_3) \\
\Gamma, \{\} & = & \Gamma \\
\{\}, \Gamma & = & \Gamma\ .
\end{array}
$$

Der *Definitionsbereich* $\mathrm{dom}(\Gamma)$ einer Typothese Γ ist die Menge aller Variablen, für die Γ Bindungen enthält:

$$
\begin{array}{rcl}
\mathrm{dom}(\{\}) & = & \{\} \\
\mathrm{dom}(x : \tau) & = & \{x\} \\
\mathrm{dom}(\Gamma_1, \Gamma_2) & = & \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)
\end{array}
$$

(CONST)	$\Gamma \vdash i : \mathsf{int}$
	$\Gamma \vdash r : \mathsf{real}$
(VAR)	$\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau \quad (x \notin dom(\Gamma_2))$
(OP)	$\dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$
	$\dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 - e_2 : \tau}$
	$\dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 * e_2 : \tau}$
	$\dfrac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 / e_2 : \tau}$
(LET)	$\dfrac{\Gamma \vdash e' : \tau' \quad \Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ e : \tau}$

Abbildung 2.2: Kontextabhängige Syntax von arithmetischen Ausdrücken

Das Regelsystem ist in Abbildung 2.2 definiert.

Man bemerke die Seitenbedingung zu Regel (VAR), die festlegt, daß tiefer geschachtelte Definitionen einer Variable äußere Definitionen derselben Variable verdecken.

Dieses Regelsystem gibt also für eine Untermenge aller Ausdrücke einen Typ an. Es legt indirekt auch eine kontext-abhängige Sprache fest, nämlich

$$\{e \mid \{\} \vdash e : \tau \text{ ist gültig für einen Typ } \tau\} \ .$$

Aufgabe 2.1 Verallgemeinern Sie die Sprache dahingehend, daß auch Operationen mit gemischten Operanden zulässig sind. Falls in einer gegebenen Typothese $e_1 : \tau_1$ und $e_2 : \tau_2$, so soll $e_1\ o\ e_2 : \tau_1 \sqcup \tau_2$ gelten, wobei o eine beliebige Binäroperation ist und

$$\tau_1 \sqcup \tau_2 \quad = \quad \begin{cases} \mathsf{int} & \text{falls } \tau_1 = \mathsf{int} \text{ und } \tau_2 = \mathsf{int}, \\ \mathsf{real} & \text{sonst.} \end{cases}$$

Welche Modifikation unseres Typsystems ist dazu nötig?

2.3 Kontextabhängige Syntax von IMP

In diesem Abschnitt definieren wir als ein längeres Beispiel die Syntax von IMP, einer einfachen imperativen Sprache.

IMP umfaßt Boolesche und ganzzahlige Ausdrücke, mit Null-Anweisung, Zuweisung, Fallunterscheidung und WHILE-Schleife als Anweisungsformen. Variablen können lokal zu einer Anweisungsfolge vereinbart werden.

Die abstrake Syntax von IMP ist wie folgt gegeben:

Variablen	x		
(ganze) Zahlen	i		
Werte	v	::=	$i \mid \mathsf{true} \mid \mathsf{false}$
Operatoren	o	::=	$+ \mid - \mid * \mid / \mid = \mid < \mid \wedge$
Ausdrücke	e	::=	$v \mid x \mid e\ o\ e \mid \neg\ e$
Anweisungen	c	::=	$\mathsf{skip} \mid x := e \mid c\ ;\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid$ $\mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{var}\ x := e\ ;\ c$

Wir legen ein abzählbar unendliches Alphabet von Variablen x zugrunde, und benutzen ganzzahlige Konstanten i wie bisher.

Diese Syntax drückt noch nicht alle Forderungen an legale IMP-Programme aus. So fehlen z. B. die Forderungen, daß

(TAUT) $\Gamma \vdash i : \mathsf{int} \qquad \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool}$

(VAR) $\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau \quad (x \notin dom(\Gamma_2))$

(IOP) $$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1\ o\ e_2 : \mathsf{int}} \ (o \in \{+, -, *, /\})$$

(CMP) $$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1\ o\ e_2 : \mathsf{bool}} \ (o \in \{<, =\})$$

(AND) $$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \wedge e_2 : \mathsf{bool}}$$

(NOT) $$\frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \neg e : \mathsf{bool}}$$

Abbildung 2.3: Kontextabhängige Syntax von Ausdrücken in IMP

- die Typen der Operanden eines Ausdrucks kompatibel zum Typ der Operation sind,
- die Bedingung einer Fallunterscheidung oder einer Schleife ein Boolescher Ausdruck ist,
- die Typen von Variable und Ausdruck in einer Zuweisung identisch sind.

Forderungen wie diese werden in einem Deduktionsystem ausgedrückt. Es gibt zwei Arten von Klauseln in diesem Deduktionssystem. Die eine Art enthält Klauseln der Form $\Gamma \vdash e : \tau$, die andere enthält Klauseln der Form $\Gamma \vdash c : \mathsf{cmd}$. Wie in der abstrakten Syntax festgelegt, ist hier e ein beliebiger Ausdruck, und c ist eine beliebige Anweisung. Γ ist in jedem Fall eine Typumgebung, d. H. eine Folge von Bindungen $x : \tau$. Wir schreiben $\Gamma.x : \tau$ für die Erweiterung von Γ um die Bindung $x : \tau$, wobei vorausgesetzt wird, daß Γ nicht schon eine Bindung für x enthält. Ein Typ τ ist schließlich ein Element von $\{\mathsf{int}, \mathsf{bool}\}$. Axiome und Regeln für Ausdrücke und Anweisungen sind in Abbildung 2.3 und Abbildung 2.4 dargestellt

Dieses Deduktionssystem definiert wie folgt eine Sprache IMP von imperativen Programmen. Sei Γ_0 eine initiale Typumgebung, die Bindungen für

(SKIP) $\Gamma \vdash \mathsf{skip} : \mathsf{cmd}$

(ASSIGN) $$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \mathsf{cmd}}$$

(SEQ) $$\frac{\Gamma \vdash c_1 : \mathsf{cmd} \quad \Gamma \vdash c_2 : \mathsf{cmd}}{\Gamma \vdash c_1\ ;\ c_2 : \mathsf{cmd}}$$

(IF) $$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash c_1 : \mathsf{cmd} \quad \Gamma \vdash c_2 : \mathsf{cmd}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \mathsf{cmd}}$$

(WHILE) $$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash c : \mathsf{cmd}}{\Gamma \vdash \mathsf{while}\ e\ \mathsf{do}\ c : \mathsf{cmd}}$$

(NEWVAR) $$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash c : \mathsf{cmd}}{\Gamma \vdash \mathsf{var}\ x := e\ ;\ c : \mathsf{cmd}}$$

Abbildung 2.4: Kontextabhängige Syntax von Anweisungen in IMP globale Variablen enthält. Dann ist ein Program p Element von IMP genau dann, wenn $\Gamma_0 \vdash p : \mathsf{cmd}$ eine gültige Aussage ist.

Beispiel 2.4 Das Program

$$p \stackrel{\text{def}}{=} \mathsf{var}\ x := 10\ ;\ \mathsf{while}\ 0 < x\ \mathsf{do}\ x := x - 1$$

ist legal (d. H. Element von IMP), da die Klausel $\Gamma_0 \vdash p : \mathsf{cmd}$ gültig ist, und zwar mit folgender Ableitung.

$$\dfrac{\Gamma_0 \vdash 10 : \mathsf{int} \qquad \dfrac{\dfrac{\Gamma_0.x : \mathsf{int} \vdash 0 : \mathsf{int} \quad \Gamma_0.x : \mathsf{int} \vdash x : \mathsf{int}}{\Gamma_0.x : \mathsf{int} \vdash 0 < x : \mathsf{bool}}\,(\mathrm{CMP}) \qquad (*)}{\Gamma_0.x : \mathsf{int} \vdash \mathsf{while}\ 0 < x\ \mathsf{do}\ x := x - 1 : \mathsf{cmd}}\,(\mathrm{WHILE})}{\Gamma_0 \vdash \mathsf{var}\ x := 10\ ;\ \mathsf{while}\ 0 < x\ \mathsf{do}\ x := x - 1 : \mathsf{cmd}}\,(\mathrm{NEWVAR})$$

wo (*) die folgende Ableitung ist:

$$\dfrac{\Gamma_0.x : \mathsf{int} \vdash x : \mathsf{int} \qquad \dfrac{\Gamma_0.x : \mathsf{int} \vdash x : \mathsf{int} \quad \Gamma_0.x : \mathsf{int} \vdash 1 : \mathsf{int}}{\Gamma_0.x : \mathsf{int} \vdash x - 1 : \mathsf{int}}\,(\mathrm{IOP})}{\Gamma_0.x : \mathsf{int} \vdash x := x - 1 : \mathsf{cmd}}\,(\mathrm{ASSIGN})$$

Man bemerke, daß gültige Programme positiv definiert werden, so daß sich Programmfehler nur im Fehlen einer Ableitung äußern. In der Spezifikation eines Übersetzers für IMP müßte eine solche Definition noch durch Angabe von Fehlermeldungen ergänzt werden.

Im folgenden benötigen wir den Begriff der *freien Variablen* eines Ausdrucks e oder einer Anweisung c. Informell gesagt ist dies die Menge aller Variablen, die in e oder c vorkommen, abzüglich aller Variablen, die lokal in c gebunden sind.

Definition 2.5 Die *freien Variablen* $\mathrm{fv}(e)$, $\mathrm{fv}(c)$ von IMP-Ausdrücken und

Anweisungen sind wie folgt definiert:

$$\begin{array}{lcl}
\mathrm{fv}(i) & = & \{\} \\
\mathrm{fv}(\mathsf{true}) & = & \{\} \\
\mathrm{fv}(\mathsf{false}) & = & \{\} \\
\mathrm{fv}(x) & = & \{x\} \\
\mathrm{fv}(e_1\ o\ e_2) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\neg e) & = & \mathrm{fv}(e) \\
\\
\mathrm{fv}(\mathsf{skip}) & = & \{\} \\
\mathrm{fv}(x := e) & = & \{x\} \cup \mathrm{fv}(e) \\
\mathrm{fv}(c_1\ ;\ c_2) & = & \mathrm{fv}(c_1) \cup \mathrm{fv}(c_2) \\
\mathrm{fv}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) & = & \mathrm{fv}(e) \cup \mathrm{fv}(c_1) \cup \mathrm{fv}(c_2) \\
\mathrm{fv}(\mathsf{while}\ e\ \mathsf{do}\ c) & = & \mathrm{fv}(e) \cup \mathrm{fv}(c) \\
\mathrm{fv}(\mathsf{var}\ x := e\ ;\ c) & = & \mathrm{fv}(e) \cup (\mathrm{fv}(c) \backslash \{x\})\ .
\end{array}$$

Aufgabe 2.2 Definiere kontextfreie und kontextabhängige Syntax von for- und repeat-Schleifen.

Kapitel 3

Operationelle Semantik

In den letzten Kapiteln über Syntax haben wir gesehen, wie eine Programmiersprache formal definiert werden kann. Durch bloße Angabe, welche Programme zu einer Programmiersprache gehören, wird allerdings noch nicht ausgesagt, welche Bedeutung die Programme einer Programmiersprache haben. Dies geschieht nun durch Festlegung einer *Semantik*.[1]

Die Bedeutung eines Programms kann auf verschiedene Arten festgelegt werden. Wir behandeln hier zunächst die *operationelle Semantik*, in der sich von der Semantik-Beschreibung auf direkte Weise eine abstrakte Maschine ableiten läßt. Allerdings ist eine solche Maschine recht weit von üblicher Computer-Hardware entfernt. Letztendlich wird nämlich nur die Beziehung zwischen Programmen, Eingaben und Resultaten in einem Deduktionssystem festlegt. Diese Stilrichtung der operationellen Semantik heißt ***natürliche Semantik*** [10]. Sie ist verwandt mit der ***strukturellen operationellen Semantik*** [15], in der ebenfalls ein Deduktionsssystem verwendet wird, das aber Zustandsübergänge einschließlich Zwischenresultaten beschreibt.

Was nun soll Semantik über ein IMP-Programm aussagen? Da IMP keine Ein- und Ausgabeanweisungen hat, nehmen wir an, daß Eingabe und Ausgabe über den globalen Speicher stattfindet. Die Bedeutung eines IMP-Programmes wird also durch die Menge der möglichen Paare von Zuständen (d. H. Speicherinhalten) am Anfang bzw. am Ende des Programms festgelegt. Diese Menge wird durch ein Deduktionssystem mit Klauseln der Form

$$S \vdash c \Downarrow S'$$

[1] Semantik kommt aus dem Griechischen und heißt dort “Bedeutung”.

bestimmt. Hier stellt S den Zustand vor Ausführung der Anweisung c dar, und S' steht für den Zustand nach Ausführung von c. Ein Zustand ist eine Funktion, die jede Variable x auf ihren Wert v abbildet, falls x in S definiert ist, und die ansonsten x auf die Konstante $\perp$ (für "undefiniert") abbildet. *Werte* v sind Ausdrücke, die nicht weiter ausgewertet werden müssen:

$$v \quad ::= \quad i \mid \mathsf{true} \mid \mathsf{false}$$

Eine wichtige Operation auf Zuständen ist die *Modifikation* $(S \mid x \mapsto v)$, die einen Zustand liefert, der überall identisch zu S ist, außer für das Argument x, das auf v abgebildet wird.

$$(S \mid x \mapsto v)(y) \quad = \quad \begin{cases} v & \text{falls } x = y \\ S(y) & \text{sonst} \end{cases}$$

Lemma 3.1 Wenn $x \neq y$, so ist

$$((S \mid x \mapsto v) \mid y \mapsto w) \quad = \quad ((S \mid y \mapsto w) \mid x \mapsto v) \ .$$

Beweis:

$$\begin{aligned} ((S \mid x \mapsto v) \mid y \mapsto w)(z) \quad &= \quad \begin{cases} v & \text{falls } z = x \\ w & \text{falls } z = y \\ S(z) & \text{sonst} \end{cases} \\ &= \quad ((S \mid y \mapsto w) \mid x \mapsto v) \end{aligned}$$

□

Ausdrücke werden gesondert behandelt, mit Klauseln der Form

$$S \vdash e \Downarrow v \ .$$

Hier steht S wiederum für den Zustand bei Auswertung von S, während der Wert v das Resultat von e darstellt.

Abbildung 3.1 behandelt die natürliche Semantik von Ausdrücken. Die (VALUE)-Regeln sagen aus, daß Werte sich zu sich selbst auswerten. Die (VAR)-Regel sagt aus, daß eine Variable x sich zu ihrem Wert $S(x)$ im gegenwärtigen Zustand S auswertet. Wenn x nicht in S definiert ist (d. H. $S(x) = \perp$), kann die Seitenbedingung $v = S(x)$ für keinen Wert v erfüllt werden, die Regel ist also in diesem Fall nicht anwendbar. In den (IOP)- und (CMP)-Regeln kommen Operatoren wie $(+)$ oder $(=)$ in zwei verschiedenen Rollen vor, einmal als Symbole der IMP-Sprache in den Ausdrücken der Schlußfolgerungen, ein anderesmal als mathematische Symbole in den

(VALUE) $S \vdash i \Downarrow i \qquad S \vdash \mathsf{false} \Downarrow \mathsf{false} \qquad S \vdash \mathsf{true} \Downarrow \mathsf{true}$

(VAR) $S \vdash x \Downarrow v \quad (S(x) = v)$

(IOP)

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 + e_2 \Downarrow v} (v = v_1 + v_2)$$

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 - e_2 \Downarrow v} (v = v_1 - v_2)$$

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 * e_2 \Downarrow v} (v = v_1 * v_2)$$

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 / e_2 \Downarrow v} (v_2 \neq 0, v = v_1 / v_2)$$

(CMP)

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 = e_2 \Downarrow \mathsf{true}} (v_1 = v_2)$$

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 = e_2 \Downarrow \mathsf{false}} (v_1 \neq v_2)$$

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 = e_2 \Downarrow \mathsf{true}} (v_1 < v_2)$$

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 < e_2 \Downarrow \mathsf{false}} (v_1 \geq v_2)$$

(NOT)

$$\frac{S \vdash e \Downarrow \mathsf{true}}{\Gamma \vdash \neg e \Downarrow \mathsf{false}} \qquad \frac{S \vdash e \Downarrow \mathsf{false}}{\Gamma \vdash \neg e \Downarrow \mathsf{true}}$$

(AND)

$$\frac{S \vdash e_1 \Downarrow \mathsf{true} \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 \wedge e_2 \Downarrow v_2} \qquad \frac{S \vdash e_1 \Downarrow \mathsf{false}}{S \vdash e_1 \wedge e_2 \Downarrow \mathsf{false}}$$

Abbildung 3.1: Natürliche Semantik von IMP-Ausdrücken

Seitenbedingungen. Der Unterschied ist wichtig — in IMP-Ausdrücken sind Operatoren reine Satzzeichen, denen Bedeutung erst noch gegeben werden muß, während in den Seitenbedingungen dieselben Operatoren ihre gewohnte mathematische Bedeutung haben. Die zweite (AND)-Regel legt fest, daß die Konjunktion *kurzgeschlossen* ausgewertet wird: Falls der erste Operand false ergibt, muß der zweite Operand nicht mehr ausgewertet werden.

Aufgabe 3.1 Die Sprache Pascal legt fest, daß immer beide Operanden einer Booleschen Binäroperation ausgewertet werden. Wie kann das mit natürlicher Semantik spezifiziert werden?

Aufgabe 3.2 Betrachte die folgende Modifikation der (AND)-Regel.

$$\text{(PAND)}\quad \frac{S \vdash e_1 \Downarrow \mathsf{true} \qquad S \vdash e_2 \Downarrow \mathsf{true}}{S \vdash e_1 \wedge e_2 \Downarrow \mathsf{true}}$$
$$\frac{S \vdash e_1 \Downarrow \mathsf{false}}{S \vdash e_1 \wedge e_2 \Downarrow \mathsf{false}} \qquad \frac{S \vdash e_2 \Downarrow \mathsf{false}}{S \vdash e_1 \wedge e_2 \Downarrow \mathsf{false}}$$

Wie muß ein Interpreter Konjunktionen auswerten, damit das von (PAND) verlangte Verhalten erzielt wird?

Abbildung 3.2 behandelt die natürliche Semantik von Anweisungen. Die (SKIP)-Regel sagt aus, daß skip den Eingangszustand unverändert läßt. Die (ASSIGN)-, (SEQ)- und (IF)-Regeln sind alle selbsterklärend. Die (WHILE)-Regel wird in zwei Fälle aufgespalten. Falls die Schleifenbedingung sich zu false auswertet, läßt die Schleife den Eingangszustand unverändert. Bei wahrer Schleifenbedingung ist die operationelle Bedeutung der Schleife gleich der Ausführung ihrer Anweisung, gefolgt vom nochmaligen Durchlauf der Schleife. Es besteht also folgender Zusammenhang:

Lemma 3.2 Für alle Zustände S, S' gilt:

$$S \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S' \tag{3.1}$$

genau dann, wenn

$$S \vdash \mathsf{if}\ e\ \mathsf{then}\ c\ ;\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip} \Downarrow S' \ . \tag{3.2}$$

Beweis: Nimm zunächst an, daß e im Zustand S keinen Wert liefert, d. H. es existiert kein Wert v, so daß $S \vdash e \Downarrow v$. Dann gibt es aber auch keinen Zustand S', für den (3.1) or (3.2) gültig ist, wie durch Inspektion

$$\text{(SKIP)}\quad S \vdash \mathsf{skip} \Downarrow S$$

$$\text{(ASSIGN)}\quad \frac{S \vdash e \Downarrow v}{S \vdash x := e \Downarrow (S \mid x \mapsto v)}$$

$$\text{(SEQ)}\quad \frac{S \vdash c_1 \Downarrow S' \qquad S' \vdash c_2 \Downarrow S''}{S \vdash c_1\ ;\ c_2 \Downarrow S''}$$

$$\text{(IF)}\quad \frac{S \vdash e \Downarrow \mathsf{true} \qquad S \vdash c_1 \Downarrow S'}{S \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \Downarrow S'}$$
$$\frac{S \vdash e \Downarrow \mathsf{false} \qquad S \vdash c_2 \Downarrow S'}{S \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \Downarrow S'}$$

$$\text{(WHILE)}\quad \frac{S \vdash e \Downarrow \mathsf{false}}{S \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S}$$
$$\frac{S \vdash e \Downarrow \mathsf{true} \qquad S \vdash c \Downarrow S' \qquad S' \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S''}{S \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S''}$$

$$\text{(NEWVAR)}\quad \frac{S \vdash e \Downarrow v \qquad (S \mid x \mapsto v) \vdash c \Downarrow S'}{S \vdash \mathsf{var}\ x := e\ ;\ c \Downarrow (S' \mid x \mapsto S(x))}$$

Abbildung 3.2: Natürliche Semantik von IMP-Anweisungen der Regeln (WHILE) und (IF) festgestellt werden kann. Also kann dieser Fall ausgeschlossen werden. Wir machen nun eine Fallunterscheidung nach dem Wert v, der von e zurückgegeben wird. Ist $v = \mathsf{false}$, so ist jeweils genau folgende Ableitung möglich:

$$\frac{S \vdash e \Downarrow \mathsf{false}}{S \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S}\ (\mathrm{WHILE})$$

bzw.

$$\frac{S \vdash e \Downarrow \mathsf{false} \qquad S \vdash \mathsf{skip} \Downarrow S}{S \vdash \mathsf{if}\ e\ \mathsf{then}\ c\ ;\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip} \Downarrow S}\ (\mathrm{IF})\ .$$

Also gilt die Äquivalenz mit $S = S'$. Ist $v = \mathsf{true}$, so ist jeweils genau folgende Ableitung möglich:

$$\frac{S \vdash e \Downarrow \mathsf{true} \qquad S \vdash c \Downarrow S' \qquad S' \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S''}{S \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S''}\ (\mathrm{WHILE})$$

bzw.

$$\frac{S \vdash e \Downarrow \mathsf{true} \qquad \dfrac{S \vdash c \Downarrow S' \qquad S' \vdash \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S''}{S \vdash c\ ;\ \mathsf{while}\ e\ \mathsf{do}\ c \Downarrow S''}\ (\mathrm{SEQ})}{S \vdash \mathsf{if}\ e\ \mathsf{then}\ c\ ;\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip} \Downarrow S''}\ (\mathrm{IF})$$

Da beide Ableitungen die gleichen Prämissen haben, ist (3.1) gültig genau dann, wenn (3.2) gültig ist. □

Man bemerke, daß eine while-Anweisung sowohl in der Schlußfolgerung wie auch in einer Prämisse der zweiten while-Regel steht. Das kann zu unendlichen Ableitungen führen, etwa für den folgenden Fall:

$$\frac{S \vdash \mathsf{true} \Downarrow \mathsf{true} \qquad S \vdash \mathsf{skip} \Downarrow S \qquad \overset{\vdots}{S \vdash \mathsf{while\ true\ do\ skip} \Downarrow S'}}{S \vdash \mathsf{while\ true\ do\ skip} \Downarrow S'}$$

Wir verlangen aber für die Gültigkeit einer Klausel einen endlichen Ableitungsbaum. $S \vdash \mathsf{while\ true\ do\ skip} \Downarrow S'$ ist also keine gültige Aussage.

Die letzte Regel der natürlichen Semantik von IMP ist (NEWVAR). Diese Regel implementiert Blockstruktur durch einen Keller für jede Variable x. Bei Ausführung von $\mathsf{var}\ x := e\ ;\ c$ wird x mit dem Resultat von e als Initialwert im Speicher eingetragen. Bei Verlassen des Sichtbarkeitsbereichs

wird der alte Wert von x zurückgespeichert. Dieses Vorgehen ist korrekt, solange auf verdeckte Instanzen von Variablen nicht indirekt, etwa über Zeiger, zugegriffen werden kann. Eine allgemeinere Lösung, die auch solche Zugriffe unterstützt, werden wir später kennenlernen.

Im vorangehenden Beweis wurden jeweils nur die letze Stufe oder die letzten zwei Stufen einer Ableitung betrachtet. Manchmal ist es damit allerdings nicht getan, sondern es muß ausgehend von den Blättern des Baumes argumentiert werden. Dazu ist folgendes Induktionsprinzip für Ableitungsbäume nützlich:

Satz 3.3 In einem Deduktionssystem sei P eine Eigenschaft von Klauseln. Wenn

- P für alle Axiome gilt, und
- für jede Ableitungsregel $\dfrac{K_1 \ldots K_n}{L}$ gilt, daß aus $P(K_1) \wedge \ldots \wedge P(K_n)$ folgt, daß P(L) gilt,

dann gilt P für alle gültigen Klauseln des Deduktionssystems.

Dieser Satz führt zu folgender Beweisstrategie: Um eine Aussage P für alle gültigen Klauseln eines Deduktionssystems nachzuweisen,

- zeige P für alle Axiome, und
- zeige P für jede Schlußfolgerung einer Inferenzregel unter der Annahme, daß P für alle Prämissen der Regel gilt.

Dies wird nun im Beweis des folgenden Satzes angewandt.

Satz 3.4 (a) Falls $x \notin \mathrm{fv}(e)$ und $S \vdash e \Downarrow v$, so gilt auch $(S \mid x \mapsto w) \vdash e \Downarrow v$.
(b) Falls $x \notin \mathrm{fv}(c)$ und $S \vdash c \Downarrow S'$, so gilt auch $(S \mid x \mapsto w) \vdash c \Downarrow (S' \mid x \mapsto w)$.

Beweis: Durch eine Induktion über den Ableitungsbaum von $S \vdash c \Downarrow S'$ bzw. $S \vdash e \Downarrow v$. Wir greifen drei Beispielfälle heraus:

Fall 1: Sei $e = y$. Dann ist die einzige anwendbare Ableitungsregel $S \vdash y \Downarrow v \quad (v = S(y))$. Da sicherlich $y \in \mathrm{fv}(y)$ und andererseits $x \notin \mathrm{fv}(y)$, muß gelten, daß $x \neq y$. Deshalb, wegen der Definition von $(S \mid x \mapsto v)$, $(S \mid x \mapsto w) \vdash e \Downarrow v$.

Fall 2: Sei $e = e_1 + e_2$, und nimm an, daß $S \vdash e \Downarrow v$. Dann ist die einzige anwendbare Ableitungsregel

$$\frac{S \vdash e_1 \Downarrow v_1 \quad S \vdash e_2 \Downarrow v_2}{S \vdash e_1 + e_2 \Downarrow v} \, (v = v_1 + v_2).$$

Per Induktionsannahme haben wir, daß $(S \mid x \mapsto w) \vdash e_1 \Downarrow v_1$ und $(S \mid x \mapsto w) \vdash e_2 \Downarrow v_2$. Also schließen wir wieder mit der (+)-Regel, daß $(S \mid x \mapsto w) \vdash e_1 + e_2 \Downarrow v$.

Die anderen Basisfälle und Induktionsschritte sind ähnlich. Der einzige Fall, der leicht anders gelagert ist, ist

Fall 3: Sei $c = \mathsf{var}\ y := e\ ;\ c'$. Dann ist die einzige anwendbare Ableitungsregel

$$\frac{S \vdash e \Downarrow v \quad (S \mid y \mapsto v) \vdash c' \Downarrow S'}{S \vdash \mathsf{var}\ y := e\ ;\ c' \Downarrow (S' \mid y \mapsto S(y))} \ .$$

Dann gilt wegen der Induktionshypothese, daß $(S \mid x \mapsto w) \vdash e \Downarrow v$. Wir machen eine Fallunterscheiung, ob $x = y$.

Ist $x = y$, so ist $((S \mid x \mapsto w) \mid y \mapsto v) = (S \mid y \mapsto v)$. Also ist

$$((S \mid x \mapsto w) \mid y \mapsto v) \vdash c' \Downarrow S'$$

als Prämisse der Ableitung gültig. Es folgt durch Anwendung der (NEWVAR)-Regel, daß

$$(S \mid x \mapsto w) \vdash \mathsf{var}\ y := e\ ;\ c' \Downarrow (S' \mid y \mapsto S(y)) \ .$$

Nun ist aber wegen $x = y$

$$((S' \mid y \mapsto S(y)) \mid x \mapsto w) \ = \ (S' \mid y \mapsto (S \mid x \mapsto w)) \ .$$

Zusammengefaßt haben wir also

$$(S \mid x \mapsto w) \vdash \mathsf{var}\ y := e\ ;\ c' \Downarrow (S' \mid y \mapsto (S \mid x \mapsto w)),$$

und damit die Behauptung.

Wenn $x \neq y$, so ist jedenfalls $x \notin \mathrm{fv}(c)$, und deshalb gilt wegen der Induktionshypothese, daß $(S \mid x \mapsto w) \vdash c' \Downarrow (S' \mid x \mapsto w)$. Also gilt mit der (NEWVAR)-Regel, daß

$$(S \mid x \mapsto w) \vdash \mathsf{var}\ y := e\ ;\ c' \Downarrow ((S' \mid x \mapsto w) \mid y \mapsto S(y)) \ .$$

Da $x \neq y$, ist aber mit Lemma 3.1

$$((S' \mid x \mapsto w) \mid y \mapsto S(y)) = ((S' \mid y \mapsto S(y)) \mid x \mapsto w) \ .$$

□

Satz 3.5 IMP ist *deterministisch*. Es gibt höchstens einen Wert v, bzw einen Zustand S' so daß $S \vdash e : v$ bzw $S \vdash c : S'$ gilt.

Aufgabe 3.3 Definiere die Semantik von `repeat` Schleifen.

Kapitel 4

Denotationelle Semantik

4.1 Einführung

In diesem Kapitel lernen wir einen anderen Definitionsstil für die Semantik von Programmen kennen. Der Stil beruht darauf, daß eine Abbildung angegeben wird, die jedem Konstrukt der Sprache seine Bedeutung (oder *Denotation*) zuordnet. Der Begriff "Bedeutung" ist hier noch offengelassen, in der Regel meint er aber bekannte Konstruktionen aus der Mathematik, z. B. Funktionen oder kartesisches Produkt. Es wird vorausgesetzt, daß die Bedeutungsabbildung *kompositionell* ist. Das heißt, die Bedeutung eines jeden Sprachkonstrukts muß sich aus den Bedeutungen seiner Teile ergeben.

Beispiel 4.1 Betrachte die natürliche Semantik von Abbildung 3.1 und Abbildung 3.2, und definiere:

- Die Bedeutung eines Ausdrucks e ist eine Funktion von Zuständen nach Werten, die einen Zustand S auf den Wert v abbildet, wenn $S \vdash e \Downarrow v$ gilt.
- Die Bedeutung einer Anweisung c ist ein *Zustandstransformer*, d.h. eine Funktion von Zuständen nach Zuständen, die einen Zustand S auf den Zustand S' abbildet, wenn $S \vdash e \Downarrow S'$ gilt.

Da IMP deterministisch ist (Satz 3.5), sind diese Abbildungen eindeutig definiert. Dennoch ist diese Definition keine denotationelle Semantik. Zum einen ist sie nicht vollständig definiert. So hat z. B. das Konstrukt

```
while true do skip
```

keine Bedeutung, da keine Klausel der Form

$$S \vdash \textsf{while true do skip} \Downarrow S'$$

gültig ist. Zum anderen ist die Semantik nicht kompositionell, da die Bedeutung der while-Schleife in der Regel (WHILE) sich widerum auf die Bedeutung der while-Schleife abstützt.

Im folgenden entwickeln wir Mechanismen, die es uns ermöglichen, eine vollständige und kompositionelle Semantik für IMP anzugeben.

Zunächst zur Vollständigkeit. Um Ausdrücken wie 1/0 oder Anweisungen wie while true do $SKIP$ eine Bedeutung geben zu können, führen wir ein neues Symbol $\perp$ in unsere Bedeutungswelt ein, das für "undefiniert" steht (englisch: "bottom"). Die Bedeutung eines Ausdrucks ist dann eine Funktion von Zuständen nach Werten oder $\perp$. Analog ist die Bedeutung einer Anweisung eine Funktion von Zuständen nach Zuständen oder $\perp$. In Symbolen:

$$\begin{array}{lcl} \llbracket e \rrbracket & : & S \to v \cup \{\perp\} \\ \llbracket c \rrbracket & : & S \to S \cup \{\perp\} \end{array}$$

Eine Aussage $L : R$ sagt hier, daß L ein Element der Menge (bzw. des Bereichs, des Typs) R ist. Auf der linken Seite steht $\llbracket \cdot \rrbracket$ für die Bedeutungsabbildung. Die *syntaktischen Klammern* $\llbracket$ und $\rrbracket$ schließen traditionell Terme der definierten Sprache ein. Wenn wir zwischen verschiedenen Bedeutungsfunktionen unterscheiden wollen, schreiben wir deren Namen vor die syntaktischen Klammern, z. B. $\mathcal{C}\llbracket c \rrbracket$. Bei nur einer Bedeutungsfunktion lassen wir den Namen weg. Auf der rechten Seite meint $S \to v \cup \{\perp\}$ eine Menge von Funktionen von Zuständen S nach Werten v oder nach $\perp$ (diese Menge wird später noch genauer charakterisiert). Statt $v \cup \{\perp\}$ schreiben wir auch einfach $v_\perp$.

Abbildung 4.1 und Abbildung 4.2 definieren die Bedeutungsfunktion $\llbracket \cdot \rrbracket$ auf Ausdrücken und Anweisungen. Man beachte, daß Symbole wie $+$ wieder in zwei verschiedenen Rollen vorkommen: Einmal in syntaktischen Klammern als Zeichen der definierten Sprache, ein andermal auf der rechten Seite von Definitionen, wo sie die üblichen mathematischen Operationen auf Zahlen usw. beschreiben. Man beachte auch, daß jede Definition vollständig sein muß. Insbesondere muß ausgesagt werden, was mit undefinierten Operanden geschieht.

Aufgabe 4.1 (Siehe auch Aufgabe 3.1) Wie muß die Semantik von Ausdrücken in Abbildung 4.1 geändert werden, um die in Pascal verwendete Konjunktion, in der immer beide Operanden ausgewertet werden, auszudrücken?

$$\begin{array}{lcl}
\llbracket i \rrbracket S & = & i \\
\llbracket \textsf{false} \rrbracket S & = & \mathit{false} \\
\llbracket \textsf{true} \rrbracket S & = & \mathit{true} \\
\llbracket x \rrbracket S & = & S(x) \\
\llbracket e_1 + e_2 \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \text{ oder } \llbracket e_2 \rrbracket S = \perp \\ \llbracket e_1 \rrbracket S + \llbracket e_2 \rrbracket S & \text{sonst} \end{cases} \\
\llbracket e_1 - e_2 \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \text{ oder } \llbracket e_2 \rrbracket S = \perp \\ \llbracket e_1 \rrbracket S - \llbracket e_2 \rrbracket S & \text{sonst} \end{cases} \\
\llbracket e_1 * e_2 \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \text{ oder } \llbracket e_2 \rrbracket S = \perp \\ \llbracket e_1 \rrbracket S * \llbracket e_2 \rrbracket S & \text{sonst} \end{cases} \\
\llbracket e_1 / e_2 \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \text{ oder } \llbracket e_2 \rrbracket S \in \{\perp, 0\} \\ \llbracket e_1 \rrbracket S / \llbracket e_2 \rrbracket S & \text{sonst} \end{cases} \\
\llbracket e_1 = e_2 \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \text{ oder } \llbracket e_2 \rrbracket S = \perp \\ \mathit{false} & \text{sonst, falls } \llbracket e_1 \rrbracket S \neq \llbracket e_2 \rrbracket S \\ \mathit{true} & \text{sonst, falls } \llbracket e_1 \rrbracket S = \llbracket e_2 \rrbracket S \end{cases} \\
\llbracket e_1 < e_2 \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \text{ oder } \llbracket e_2 \rrbracket S = \perp \\ \mathit{false} & \text{sonst, falls } \llbracket e_1 \rrbracket S \geq \llbracket e_2 \rrbracket S \\ \mathit{true} & \text{sonst, falls } \llbracket e_1 \rrbracket S < \llbracket e_2 \rrbracket S \end{cases} \\
\llbracket e_1 \wedge e_2 \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \\ \mathit{false} & \text{falls } \llbracket e_1 \rrbracket S = \mathit{false} \\ \llbracket e_2 \rrbracket S & \text{sonst} \end{cases} \\
\llbracket \neg e \rrbracket S & = & \begin{cases} \perp & \text{falls } \llbracket e_1 \rrbracket S = \perp \\ \mathit{false} & \text{falls } \llbracket e_1 \rrbracket S = \mathit{true} \\ \mathit{true} & \text{falls } \llbracket e_1 \rrbracket S = \mathit{false} \end{cases}
\end{array}$$

Abbildung 4.1: Denotationelle Semantik von IMP Ausdrücken

$$
\begin{aligned}
\mathcal{C}[\![\mathsf{skip}]\!]S &= S \\
\mathcal{C}[\![x := e]\!]S &= \begin{cases} \bot & \text{falls } [\![e]\!]S = \bot \\ (S \mid x \mapsto [\![e]\!]S) & \text{sonst} \end{cases} \\
\mathcal{C}[\![c_1 \,;\, c_2]\!]S &= (\mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!])S \\
\mathcal{C}[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]S &= \begin{cases} \bot & \text{falls } [\![e]\!]S = \bot \\ \mathcal{C}[\![c_1]\!]S & \text{falls } [\![e]\!]S = \mathit{true} \\ \mathcal{C}[\![c_2]\!]S & \text{falls } [\![e]\!]S = \mathit{false} \end{cases} \\
\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]S &= \mathsf{fix}\ (\mathit{unroll})\ (S) \\
\text{wobei } \mathit{unroll}\ (\mathit{loop})\ (S) &= \begin{cases} \bot & \text{falls } [\![e]\!]S = \bot \\ (\mathit{loop} \circ \mathcal{C}[\![c]\!])(S) & \text{falls } [\![e]\!]S = \mathit{true} \\ S & \text{falls } [\![e]\!]S = \mathit{false} \end{cases} \\
\mathcal{C}[\![\mathsf{var}\ x := e \,;\, c]\!]S &= \begin{cases} \bot & \text{falls } [\![e]\!]S = \bot \\ (\mathcal{C}[\![c]\!](S \mid x \mapsto [\![e]\!]S) \mid x \mapsto S(x)) & \text{sonst} \end{cases}
\end{aligned}
$$

Hilfsdefinition:

$$
(f \circ g)\ (x) = \begin{cases} \bot & \text{falls } g(x) = \bot \\ f(g(x)) & \text{sonst} \end{cases}
$$

Abbildung 4.2: Denotationelle Semantik von IMP Anweisungen

Der kritische Punkt dieser Definition liegt in der Definition der Bedeutung der while-Schleife. Die Bedeutung von whileist gegeben als der Fixpunkt der *unroll*-Funktion.

Definition. Ein Wert x ist Fixpunkt einer Funktion f falls $f(x) = x$.

unroll ist eine Funktion von Zustandstransformern nach Zustandstransformern:

$$\mathit{unroll} \quad : \quad (S \rightarrow S) \rightarrow (S \rightarrow S)$$

Informell gesagt ist *unroll* die Funktion, die eine while-Schleife einmal ausrollt, d. H. für eine Schleife while e do c ist

$$\mathit{unroll}(\mathit{loop}) \quad = \quad \mathcal{C}[\![\mathsf{if}\ e\ \mathsf{then}\ c\ ;\ \mathit{loop}\ \mathsf{else\ skip}]\!].$$

Entsprechend erklärt sich die Wirkung einer n-maligen Anwendung von *unroll*:

$$
\begin{array}{lcl}
\mathit{unroll}^n(\mathit{loop}) & = & \mathcal{C}[\![\mathsf{if}\ e\ \mathsf{then}\ c\ ; \\
& & \quad \mathsf{if}\ e\ \mathsf{then}\ c\ ; \\
& & \quad\quad \ldots \\
& & \quad\quad\quad \mathsf{if}\ e\ \mathsf{then}\ c\ ;\ \mathit{loop} \\
& & \quad\quad\quad \mathsf{else\ skip} \\
& & \quad\quad \ldots \\
& & \quad \mathsf{else\ skip} \\
& & \mathsf{else\ skip} \\
& &]\!]
\end{array}
$$

Die Wirkung einer while-Schleife kann immer durch die Wirkung von if-then-elseAnweisungen erklärt werden, falls die Schleife "genügend oft" ausgerollt wird (nämlich mindestens einmal pro Durchlauf der Schleife). Das Problem ist, daß statisch nicht vorausgesagt werden kann, wie oft eine while-Schleife durchlaufen wird, also ist auch die benötigte Anzahl der Ausrollungen unbekannt. Deshalb nun die indirekte Definition mittels des Fixpunktes. Im Prinzip wird gesagt, daß eine while-Schleife so oft ausgerollt wird, bis sich an der Bedeutung der Schleife nichts mehr ändert.

Beispiel 4.2 Betrachte die Schleife while $x \neq 0$ do $x := x - 1$. Dann ist

$$
\mathit{unroll}(\mathit{loop})(S) \quad = \quad \begin{cases} S & \text{falls } S(x) = 0, \\ \mathit{loop}(S \mid x \mapsto S(x) - 1) & \text{falls } S(x) \neq 0. \end{cases}
$$

Ein Fixpunkt von *unroll* ist die Funktion

$$
f(S) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{falls } S(x) < 0, \\ (S \mid x \mapsto 0) & \text{falls } S(x) \geq 0. \end{cases}
$$

Dies wird wie folgt nachgeprüft:

$$\begin{aligned}
unroll(f)(S) &= \begin{cases} f(S \mid x \mapsto S(x)-1) & \text{falls } S(x) < 0, \\ S & \text{falls } S(x) = 0, \\ f(S \mid x \mapsto S(x)-1) & \text{falls } S(x) > 0. \end{cases} \\
&= \begin{cases} \bot & \text{falls } S(x) < 0, \\ S & \text{falls } S(x) = 0, \\ (S \mid x \mapsto 0) & \text{falls } S(x) > 0. \end{cases} \\
&= \begin{cases} \bot & \text{falls } S(x) < 0, \\ (S \mid x \mapsto 0) & \text{falls } S(x) \geq 0. \end{cases} \\
&= f(S)
\end{aligned}$$

Da S beliebig war, gilt $unroll(f) = f$, also ist f Fixpunkt von $unroll$. Damit ist f auch die Bedeutung der while-Schleife, was unserem operationellen Verständis entspricht.

Im allgemeinen wirft aber die Definition über Fixpunkte Probleme auf, die noch zu lösen sind. Nämlich:

- Nicht jede Funktion hat einen Fixpunkt. Sei z. B. $succ$ die Funktion, die jede natürliche Zahl auf ihren Nachfolger abbildet. Dann existiert keine natürliche Zahl x für die $x = succ(x)$ gilt.

- Manche Funktionen haben mehr als einen Fixpunkt. Beispielsweise hat die Identitätsfunktion auf natürlichen Zahlen jede natürliche Zahl als Fixpunkt.

Der nächste Abschnitt beschreibt Lösungen für diese beiden Probleme. Wir lösen das Existenzproblem, indem wir Bedeutungen nur in bestimmten mathematischen Strukturen, den vollständigen Halbordnungen mit stetigen Funktionen, definieren. Und wir lösen das Eindeutigkeitsproblem, indem wir den gewünschten Fixpunkt zusätzlich charakterisieren.

4.2 Vollständige Halbordnungen

Definition. Eine *Halbordnung* ist ein Paar $(D, \sqsubseteq)$, bestehend aus einer Menge D und einer Relation $\sqsubseteq$ auf $D \times D$, so daß für alle $x, y, z \in D$ gilt:

$$\begin{array}{llll}
x \sqsubseteq x & & & \text{(Reflexivität)} \\
x \sqsubseteq y, y \sqsubseteq x & \Rightarrow & x = y & \text{(Anti-Symmetrie)} \\
x \sqsubseteq y, y \sqsubseteq z & \Rightarrow & x \sqsubseteq z & \text{(Transitivität)}
\end{array}$$

Definition. Ein Wert $x \in D$ ist *obere Schranke* einer Menge $M \subseteq D$, falls für alle $y \in M$ gilt, daß $y \sqsubseteq x$. Wir schreiben dann auch $M \sqsubseteq x$. Ein *Supremum* ist eine kleinste obere Schranke. x ist also Supremum von M genau dann, wenn $M \sqsubseteq x$ und $\forall y \in D.\ M \sqsubseteq y \Rightarrow x \sqsubseteq y$. Wir schreiben auch $\bigsqcup M$ für das Supremum von M, falls es existiert.

Satz 4.3 Jede Menge hat höchstens ein Supremum.

Beweis: Seinen x und y Suprema von M. Dann ist $x \sqsubseteq y$ und $y \sqsubseteq x$. Also ist $x = y$ wegen der Anti-Symmetrie von $\sqsubseteq$. □

Definition. Eine Halbordung $(D, \sqsubseteq)$ ist *vollständig*, falls jede aufsteigende Folge in D ein Supremum hat.

Für vollständige Halbordnung verwenden wir auch die Abkürzung *CPO* (von englisch *Complete Partial Order*).

Satz 4.4 Jede *CPO* $(D, \sqsubseteq)$ hat ein kleinstes Element.

Beweis: Die leere Folge $\{\}$ ist aufsteigend, und hat deshalb ein Supremum, $\bigsqcup\{\}$. Sei $x \in D$ beliebig. Trivialerweise gilt $\{\} \sqsubseteq x$. Also folgt $\bigsqcup\{\} \sqsubseteq x$. □

Im folgenden verwenden wir immer das Symbol $\bot$ für das kleinste Element einer *CPO*.

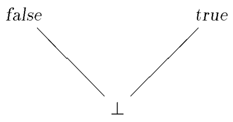
Beispiel 4.5 Beispiele für *CPO*s.

1. Die Potenzmenge $\wp(M)$ einer Menge M mit $\{\}$ als kleinstem Element und $\subseteq$ als $\sqsubseteq$.

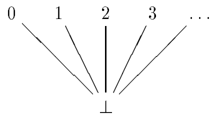
2. Die ein-elementige *CPO* $\{\bot\}$.

3. Die zwei-elementige *CPO* $\mathbf{O} = \{\bot, \top\}$ mit $\bot \sqsubseteq \top$.

4. Die *CPO* **Bool** der Wahrheitwerte.



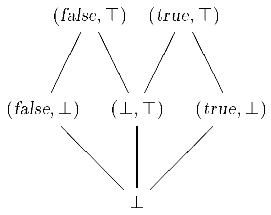
5. Die *CPO* der natürlichen Zahlen, **Nat**.



O, **Bool** und **Nat** sind Beispiele für *flache CPO*s, in denen alle Ordnungsbeziehungen von der Form $x \sqsubseteq x$ oder $\perp \sqsubseteq x$ sind. Ein Beispiel für eine nicht-flache *CPO* ist:

6. Die Menge der natürlichen Zahlen mit der gewöhnlichen Ordnung und ω als größtem Element.

7. Die *CPO* der Paare mit erstem Element aus **Bool** und zweitem Element aus **O**. Diese *CPO* wird auch das *kartesische Produkt* aus **Bool** und **O** genannt, in Symbolen **Bool** $\times$ **O**.



4.3 Funktionen auf vollständigen Halbordnungen

Definition. Seien D und E *CPO*s. Eine Funktion f von D nach E ist *monoton*, wenn für alle $x \sqsubseteq y$ gilt, daß $f(x) \sqsubseteq f(y)$. f ist *stetig*, wenn für jede aufsteigende Folge $(x_i)_{i \in I}$ gilt:

$$\bigsqcup_{i \in I} f(x_i) \quad = \quad f(\bigsqcup_{i \in I} x_i).$$

Man bemerke den Bezug zur Folgen-Stetigkeit: Eine Funktion ist (folgen-) stetig, wenn für jede konvergierende Folge $(x_i)_{i \in I}$ gilt:

$$\lim_{i \in I} f(x_i) \quad = \quad f(\lim_{i \in I} x_i).$$

Man gelangt also von der Folgen-Stetigkeit zur CPO-Stetigkeit, wenn man "konvergierend" durch "aufsteigend" und "lim" durch "$\bigsqcup$" ersetzt.

Satz 4.6 Jede stetige Funktion ist monoton.

Beweis: Sei f stetig und sei $x \sqsubseteq y$. Betrachte die zwei-elementige Folge x, y. Da f stetig ist, ist $\bigsqcup\{f(x), f(y)\} = f(\bigsqcup\{x, y\}) = f(y)$. Nun ist aber sicher $f(x) \sqsubseteq \bigsqcup\{f(x), f(y)\}$. Also folgt $f(x) \sqsubseteq f(y)$. □

Eine wichtige Eigenschaft von stetigen Funktionen zwischen *CPO*s ist, daß sie Fixpunkte besitzen, wie im folgenden *Fixpunktsatz* gezeigt wird.

Satz 4.7 Sei D eine *CPO* und sei $f : D \to D$ stetig. Dann hat f mindestens einen Fixpunkt, und $\mathsf{fix}(f) \stackrel{\text{def}}{=} \bigsqcup_{n=0,1,\ldots} f^n(\perp)$ ist kleinster Fixpunkt von f.

Beweis: Wir zeigen zunnächst, daß $\mathsf{fix}(f)$ Fixpunkt von f ist.

$$\begin{array}{rcll} f(\mathsf{fix}(f)) & = & f(\bigsqcup_{n=0,1,\ldots} f^n(\bot)) & \text{nach Def. von fix} \\ & = & \bigsqcup_{n=0,1,\ldots} f(f^n(\bot)) & \text{da } f \text{ stetig ist} \\ & = & \bigsqcup_{n=1,2,\ldots} f^n(\bot) & \\ & = & \bot \sqcup \bigsqcup_{n=1,2,\ldots} f^n(\bot) & \text{da } \bot \text{ kleinstes Element ist} \\ & = & \bigsqcup_{n=0,1,\ldots} f^n(\bot) & \\ & = & \mathsf{fix}(f) & \text{nach Def. von fix} \end{array}$$

Wir zeigen nun, daß $\mathsf{fix}(f)$ kleinster Fixpunkt von f ist. Sei x ein Fixpunkt von f. Wir zeigen mit einer Induktion über n, daß $f^n(\bot) \sqsubseteq x$, für alle n. Falls $n = 0$, haben wir $f^0(\bot) = \bot \sqsubseteq x$, da $\bot$ kleinstes Element von D ist. Für $n > 0$ gilt:

$$\begin{array}{rcll} f^n(\bot) & = & f(f^{n-1}(\bot)) & \\ & \sqsubseteq & f(x) & \text{wegen Induktionshypothese} \\ & & & \text{und Monotonie von } f \\ & = & x & \text{da } x \text{ Fixpunkt von } f \end{array}$$

Also ist x obere Schranke von $(f^n(\bot))_{n=0,1,\ldots}$. Es folgt aus der Supremumeigenschaft, daß

$$\mathsf{fix}(f) = \bigsqcup_{n=0,1,\ldots} f^n(\bot) \sqsubseteq x.$$

□

Definition. Sind D und E *CPO*s, so bezeichne $D \rightarrow E$ die Menge aller stetigen Funktionen zwischen D und E mit der punktweisen Ordnung. Das heißt für $f, g \in D \rightarrow E$ gilt $f \sqsubseteq g$ genau dann, wenn $f(x) \sqsubseteq g(x)$ für alle $x \in D$.

m Stetige Funktionen zwischen zwei *CPO*s bilden wieder eine *CPO*, wie im folgenden Satz gezeigt wird.

Satz 4.8 Seien D und E zwei *CPO*s. Dann ist $D \rightarrow E$ eine CPO.

Beweis: Sei $(f_i)_{i \in I}$ eine aufsteigende Folge in $D \rightarrow E$. Zu zeigen ist, daß $(f_i)_{i \in I}$ ein Supremum besitzt. Wegen der Definition von $\sqsubseteq$ auf Funktionen ist für jedes $x \in D$ $(f_i(x))_{i \in I}$ eine aufsteigende Folge in E. Da E eine *CPO* ist, besitzt $(f_i(x))_{i \in I}$ ein Supremum, $\bigsqcup_{i \in I} f_i(x)$. Sei f diejenige Funktion, die jedes $x \in D$ auf $\bigsqcup_{i \in I} f_i(x)$ abbildet. Dann ist f Supremum von $(f_i)_{i \in I}$,

vorausgesetzt, daß f ein Element von $D \rightarrow E$ ist. Es bleibt also zu zeigen, daß f stetig ist. Sei $(x_j)_{j \in J}$ eine aufsteigende Folge in D. Dann ist

$$\begin{array}{rcll} f(\bigsqcup_{j \in J} x_j) & = & \bigsqcup_{i \in I} f_i(\bigsqcup_{j \in J} x_j) & \text{nach Def. von } f \\ & = & \bigsqcup_{i \in I} \bigsqcup_{j \in J} f_i(x_j) & \text{da } f_i \text{ stetig, für alle } i \in I \\ & = & \bigsqcup_{j \in J} \bigsqcup_{i \in I} f_i(x_j) & \\ & = & \bigsqcup_{j \in J} f(x_j) & \end{array}$$

□

Kapitel 5

Axiomatische Semantik

In diesem Kapitel lernen wir Techniken zur axiomatischen Verifizierung von Programmen kennen.

5.1 Hoare-Logik

Wir beschreiben hier ein Deduktionssystem von Aussagen der Form

$$\{P\}\ c\ \{Q\},$$

die auch *Hoare-Tripel* genannt werden. Hier ist c eine Anweisung, und P, Q sind Prädikate, die die *Vorbedingung* und *Nachbedingung* der Anweisung festlegen. Dabei sind zwei Variationen möglich, je nachdem, ob verlangt wird, daß c terminiert.

In einem Deduktionsystem für *partielle Korrektheit* bedeutet die Aussage $\{P\}\ c\ \{Q\}$, daß die Ausführung von c in einem Zustand, der P erfüllt, entweder nicht terminiert oder einen Zustand liefert, der Q erfüllt. In einem Deduktionsystem für *totale Korrektheit* bedeutet die Aussage $\{P\}\ c\ \{Q\}$ dagegen, daß die Ausführung von c in einem Zustand, der P erfüllt, sicher terminiert und einen Zustand liefert, der Q erfüllt.

Wir werden zunächst partielle Korrektheit behandeln, da sie mit einfacheren Regeln beschreibbar ist, und totale Korrektheit im nächsten Abschnitt studieren.

Eine weitere Frage betrifft die Bedeutung der Prädikate P, Q. Der *intensionale* Ansatz faßt Prädikate einfach als Formeln “mit den üblichen (assign) $\{[e/x]P\}\ x := e\ \{P\}$

(skip) $\{P\}\ \mathsf{skip}\ \{P\}$

(seq) $$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\ ;\ c_2\ \{R\}}$$

(if) $$\frac{\{e \wedge P\}\ c_1\ \{Q\} \qquad \{\neg e \wedge P\}\ c_2\ \{Q\}}{\{P\}\ \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{Q\}}$$

(while) $$\frac{\{e \wedge P\}\ c\ \{P\}}{\{P\}\ \mathsf{while}\ e\ \mathsf{do}\ c\ \{\neg e \wedge P\}}$$

(var) $$\frac{\{x = e \wedge P\}\ c\ \{Q\}}{\{P\}\ \mathsf{var}\ x := e\ ;\ c\ \{Q\}}\ (x \notin \mathrm{fv}(P, Q))$$

(cons) $$\frac{\{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}\ (P \Rightarrow P'\ \wedge\ Q' \Rightarrow Q)$$

Hier ist $[e/x]\ P$ die textuelle Substitution aller Vorkommnisse von x in P durch e.

Abbildung 5.1: Hoare-Tripel für IMP

Rechenregeln" auf. Dabei muß strikt gesagt natürlich die Sprache solcher Formeln festgelegt werden, allerdings wird dies der Einfachheit halber oft implizit angenommen. Der *extensionale Ansatz* faßt Prädikate als Bool'sche Funktionen über Zuständen auf:

$$P = S \rightarrow \mathsf{bool}$$

Der intensionale Ansatz läßt einfachere Beweismanipulationen zu, aber um seine Korrektheit zu beweisen, muß seine Äquivalenz zur extensionalen Sicht erst noch gezeigt werden. Wir werden zunächst intensional vorgehen. Das Deduktionsystem für Hoare-Tripel ist in Abbildung 5.1 wiedergegeben.

Man bemerke, daß in der (while)-Regel das Prädikat P vor und nach der Ausführung der Schleifenanweisung c auftaucht. Dieses Prädikat wird *Schleifeninvariante* genannt.

Hoare-Tripel eignen sich besonders gut zum Programmbeweis, da Anweisungen und Prädikate alternierend geschrieben werden können.

$$\{P_1\}\ c_1\ \{P_2\}\ c_2\ \{P_3\} \ldots$$

Dabei ist mittels des Deduktionssystems zu prüfen, ob jede der Aussagen $\{P_i\}\ c_i\ \{P_{i+1}\}$ gültig ist.

Beispiel 5.1 Ein Programm zur binären Suche samt Korrektheitsbeweis.

Seien Variablen $a : \mathsf{array}\ 0..N-1\ \mathsf{of\ int}$, $n : \mathsf{int}$, $x : \mathsf{int}$ global gegeben. Wir nehmen an, daß a sortiert ist. Das folgende Programm setzt n auf einen Wert i, so daß $a[i] = x$, und setzt n auf -1, falls x in a nicht vorkommt. Wir benutzen die Invariante

$$\begin{array}{rcl} I & \stackrel{\text{def}}{=} & sorted(a) \\ & \wedge & 0 \leq l \leq r \leq N \\ & \wedge & \forall i.0 \leq i < l.a[i] \leq x \\ & \wedge & \forall i.r \leq i < N.x < a[i]. \end{array}$$

var $l := 0$;	
var $r := N$;	$\{I\}$
while $l \neq r$ do	$\{I \wedge l \neq r\}$
($m := (l + r)$ div 2	$\{I \wedge l \leq m < r\}$
if $x < a[m]$ then	$\{I \wedge l \leq m < r \wedge x < a[m]\}$
$r := m$	$\{I\}$
else	$\{I \wedge l \leq m < r \wedge x \geq a[m]\}$
$l := m + 1$	$\{I\}$
)	$\{I \wedge l = r\}$
	$=$
	$\{sorted(a) \wedge$
	$0 \leq l \leq N \wedge$
	$\forall i.0 \leq i < l.a[i] \leq x \wedge$
	$\forall i.l \leq i < N.x < a[i]\}$
if $a[l-1] = x$ then	
$n := l - 1$	$\{a[n] = x\}$
else	
$n := -1$	$\{n = -1 \ \wedge\ \forall i.0 \leq i < N.a[i] \neq x\}$

Der Programmbeweis setzt voraus, daß das Programm terminiert. Dies kann durch Angabe einer Funktion auf Zuständen, die bei jedem Durchlauf einer Schleife strikt abnimmt und deren Wertebereich nach unten begrenzt ist, geschehen. In unserem Beispiel ist $r - l$ eine solche Funktion.

Aufgabe 5.1 Gibt es eine Anweisung c, so daß $\{false\}\ c\ \{true\}$ gültig ist?

5.2 Prädikaten-Transformer

Im letzten Abschnitt haben wir gesehen, daß sich axiomatische Semantik durch ein Deduktionssystem ausdrücken läßt. In diesem Abschnitt zeigen wir, daß sich axiomatische Semantik auch als Spielart der denotationellen Semantik auffassen läßt. Die Bedeutung einer Anweisung c ist hier ein *Prädikatentransformer* $\mathrm{WP}[\![c]\!]$, der, gegeben eine Nachbedingung Q, die schwächste Vorbedingung P liefert, mit der $\{P\}c\{Q\}$ gültig ist. Um den Bedeutungsbereichen die Struktur von vollständigen Halbordungen zu geben, ist es günstig, extensional vorzugehen, d. H. ein Prädikat nicht als Formel, sondern als Funktion auf Zuständen aufzufassen. Wir gehen also

Bereiche:

$$\begin{array}{lcl} \text{Zustand} & S & = & x \to v \\ \text{Prädikat} & P & = & S \to \mathsf{bool} \end{array}$$

Semantische Abbildung:

$$\begin{array}{lcl}
\mathrm{WP}[\![c]\!] & : & P \to P \\[1ex]
\mathrm{WP}[\![\mathsf{skip}]\!]P & = & P \\
\mathrm{WP}[\![x := e]\!]P & = & P[x := [\![e]\!]] \\
\mathrm{WP}[\![c_1\ ;\ c_2]\!]P & = & \mathrm{WP}[\![c_1]\!](\mathrm{WP}[\![c_2]\!]P) \\
\mathrm{WP}[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]P & = & [\![e]\!] \wedge \mathrm{WP}[\![c_1]\!]P\ \vee\ \neg[\![e]\!] \wedge \mathrm{WP}[\![c_2]\!]P \\
\mathrm{WP}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]P & = & \mathsf{fix}(unroll)(P) \\
\text{wobei } unroll(loop)(P) & = & [\![e]\!] \wedge loop(\mathrm{WP}[\![c]\!]P)\ \vee\ \neg[\![e]\!] \wedge P \\
\mathrm{WP}[\![\mathsf{var}\ x := e\ ;\ c]\!](P[x := E]) & = & (\mathrm{WP}[\![x := e\ ;\ c]\!]P)[x := E]
\end{array}$$

Hilfsfunktionen:

$$\begin{array}{lcl}
(P \wedge Q)(S) & = & P(S) \wedge Q(S) \\
(P \vee Q)(S) & = & P(S) \vee Q(S) \\
(\neg P)(S) & = & \neg P(S)
\end{array}$$

Abbildung 5.2: Prädikatentransformer — extensionale Version

von folgenden Bereichen aus.

$$\begin{array}{lcl} \text{Prädikate} & P, Q & = & S \to \mathsf{bool} \\ \text{Transformer} & T & = & P \to P \end{array}$$

Der Kürze halber schreiben wir *true* für das Prädikat, das jeden Zustand S nach *true* abbildet, und gehen analog für *false* vor.

Prädikate werden als Halbordung mit *false* als kleinstem Element und $\Rightarrow$ als $\sqsubseteq$-Operation aufgefaßt. Für Prädikatentransformer fordern wir, daß sie stetig sind. Es kann mit Satz 4.8 gezeigt werden, daß Prädikate und Prädikatentransformer beide vollständige Halbordungen bilden, Fixpunkte von stetigen Funktionen über diesen Mengen also immer existieren.

Abbildung 5.2 definiert eine Bedeutungsfunktion, die Anweisungen auf Prädikatentransformer abbildet. In der dritten und der letzten Regel ist dabei $P[x := E]$ dasjenige Prädikat Q, für das gilt, daß $Q(S) = P(S \mid x \mapsto E(S))$ für jeden Zustand S. Man bemerke, daß bei einer extensionalen Definition von Prädikaten eine Substitution $[e/x]P$ sinnlos ist — P ist schließlich keine Formel, in der textuell ersetzt werden könnte. Die Definition von $P[x := E]$ macht es recht schwierig, mit Prädikaten zu rechnen. Allerdings kann für eingeschränkte Formel-Sprachen von Prädikaten (die z. B. alle Beispiele in diesem Kapitel abdecken) folgender Zusammenhang zwischen $P[x := E]$ und $[E/x]P$ bewiesen werden:

Sei $[\![P]\!]$ die Bedeutung der Formel P. Also:

$$[\![P]\!](S) = \mathit{true} \quad \text{genau dann, wenn} \quad P \text{ im Zustand } S \text{ gilt.}$$

Dann ist

$$[\![P]\!][x := e] \quad = \quad [\![[e/x]P]\!].$$

Kann dieser Zusammenhang bewiesen werden, so kann also statt der Ersetzung $P[x := e]$ genausogut die Substitution $[e/x]P$ verwendet werden, was berechnungstechnisch wesentlich günstiger ist. Die Semantik kann dann auch ausschließlich intensional formuliert werden, wie es in Abbildung 5.3 gezeigt ist.

Bereiche:

Prädikat P = "Boolesche Formel mit den üblichen Rechelregeln"

Semantische Abbildung:

$$
\begin{array}{lcl}
\mathrm{wp}[\![c]\!] & : & P \to P \\
\\
\mathrm{wp}[\![\mathsf{skip}]\!]P & = & P \\
\mathrm{wp}[\![x := e]\!]P & = & [e/x]P \\
\mathrm{wp}[\![c_1 \,;\, c_2]\!]P & = & \mathrm{wp}[\![c_1]\!](\mathrm{wp}[\![c_2]\!]P) \\
\mathrm{wp}[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]P & = & e \wedge \mathrm{wp}[\![c_1]\!]P \ \vee\ \neg e \wedge \mathrm{wp}[\![c_2]\!]P \\
\mathrm{wp}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]P & = & \mathsf{fix}(\mathit{unroll})(P) \\
\text{wobei } \mathit{unroll}(\mathit{loop})(P) & = & e \wedge \mathit{loop}(\mathrm{wp}[\![c]\!]P) \ \vee\ \neg e \wedge P \\
\mathrm{wp}[\![\mathsf{var}\ x := e \,;\, c]\!]([e'/x]P) & = & [e'/x](\mathrm{wp}[\![x := e \,;\, c]\!]P)
\end{array}
$$

Abbildung 5.3: Prädikatentransformer – intensionale Version

Teil II

Funktionen

Kapitel 6

Funktionales Programmieren

Ein rein funktionales Programm besteht aus Daten, Funktionen, und einem Ausdruck, der das Resultat des Programms angibt. Es fehlen dagegen Variablen und Anweisungen.

Die Auseinandersetzung mit dem funktionalen Programmieren (FP) ist aus mehreren Gründen interessant:

- Einerseits ist FP sehr einfach. Dadurch wird ein vertieftes Verständnis des Programms bzw. der Programmiersprache ermöglicht.
- Im FP ist nur der *Wert* eines Ausdrucks wichtig, Seiteneffekte können nicht auftauchen. Diese Eigenschaft wird "referentielle Transparenz" genannt. Sie erlaubt es, Programme mit besonders einfachen und mächtigen Umformungsregeln zu manipulieren.
- FP stützt sich auf ausgiebig studierte mathematische Theorien ab, nämlich das Lambda-Kalkül auf der syntaktischen Seite und die Bereichstheorie auf der semantischen Seite.
- Wegen seiner Einfachheit treten viele Konzepte im FP in klarerer Form auf als anderswo.
- Zwei typisch funktionale Konzepte, nämlich Funktionen höherer Ordnung und Auswertung bei Bedarf, erlauben eine sehr flexible Modularisierung [8].

6.1 Grundzüge

Im folgenden erläutern wir die Grundzüge des funktionalen Programmierens anhand einer einfachen Sprache, die an Haskell [7] orientiert ist.

Ein Programm in dieser Sprache besteht aus Daten, Funktionsdefinitionen, und einem Resultatsausdruck.

6.1.1 Datentypen

Datentypen sind entweder vordefiniert (z. B. Zahlen vom Typ int, Buchstaben vom Typ Char, Wahrheitswerte True, False vom Typ bool) oder sie sind *algebraische Datentypen*, die vom Programmierer definiert sind. Zwei Beispiele sind:

```
data Color   = Red | Green | Blue
data IntTree = E
             | B Int IntTree IntTree
```

Die erste Vereinbarung definiert einen Typ Color mit drei Werten, Red, Green und Blue. Die zweite Vereinbarung definiert einen Typ IntTree, dessen Werte folgendermaßen strukturiert sind:

- E ist ein Wert vom Typ IntTree.
- Wenn x, l und r drei Werte sind, so daß x:Int, l:IntTree und r:IntTree, so ist B x l r wieder ein Wert vom Typ IntTree.
- Kein anderer Wert ist vom Typ IntTree.

Wir sehen also, daß Werte algebraischer Datentypen durch einen *Konstruktor* und möglicherweise noch aus Komponenten-Werten nach dem Konstruktor bestehen. Um beide auseinanderzuhalten, schreiben wir hier den Konstruktor in Fettdruck. In Haskell werden Konstruktoren statt dessen immer großgeschrieben.

Die allgemeine Syntax einer Datentyp-Definition ist also

$$\begin{array}{lcl} \mathit{dataDcl} & = & \mathsf{data}\ \mathit{ident} = \mathit{dataType} \\ \mathit{dataType} & = & \mathit{alternative} \mid \ldots \mid \mathit{alternative} \\ \mathit{alternative} & = & \mathit{constructor}\ \mathit{type}\ \ldots\ \mathit{type} \\ \mathit{constructor} & = & \mathit{ident} \end{array}$$

Auch vordefinierte Typen können als algebraische Datentypen gesehen werden:

```
data Bool = True | False
data Char = chr(0) | ...| chr(255)
data Int  = minint | ...| maxint
```

6.1.2 Funktionen

Eine Funktion ist durch eine oderer mehrere definierende Gleichungen gegeben. Beispiel:

```
not True  = False
not False = True
```

Bei der Funktionsanwendung wird jeweils die Gleichung herangezogen, die auf das aktuelle Argument der Funktion paßt. Alternativ kann not auch mit einer Gleichung, dafür aber mit einem case- Ausdruck geschrieben werden:

```
not x = case x of
          True ⇒ False
          False ⇒ True
```

Neben den definierenden Gleichungen kann auch noch der Typ von not explizit deklariert werden:

```
not : Bool → Bool
```

Dies sagt aus, daß not eine Funktion von Bool nach Bool ist. Der →-Operator bezeichnet Funktionen. Hier weichen wir in unserer Notation von Haskell ab, das (::) für die Typisierung verwendet und (:) für Listen-Konstruktion reserviert. Wir verwenden (:) für beides, da in unseren Beispielen Verwechslungen ausgeschlossen sind.

Hier ist eine andere Funktionsdefinition:

```
flip : IntTree → IntTree

flip E         = E
flip (B x l r) = B x (flip r) (flip l)
```

Diesmal ist der formale Parameter B x l r ein *Muster*, das aus einem Konstruktor B und Variablen x, l und r besteht. Beim Funktionsaufruf wird wieder die Gleichung ausgewählt, die auf das Argument paßt (wenn mehrere passen, die erste). Zusätzlich werden die Komponenten des Arguments in die korresponierenden Variablen übertragen. Beispiel:

```
flip (B 1 (B 2 E E) E) = B 1 E (B 2 E E).
```

Neben Funktionen können natürlich auch Daten-Werte definiert werden. Beispiel:

```
oneTree = B 1 E E.
```

Um Programme besser strukturieren zu können, lassen wir noch lokale Definitionen von Funktionen oder Werten in einem let-Konstrukt zu. Beispiel:

```
let oneTree = B 1 E E
in  B 2 oneTree oneTree
```

Dies ist im wesentlichen alles! Wir werden noch vereinzelt neue Notationen kennenlernen, die die Schreibweise vereinfachen, aber alle grundlegenden Konzepte einer funktionalen Sprache wurden hier schon vorgestellt.

6.2 FP und abstrakte Datentypen

Viele abstrakte Datentypen haben eine direkte Implementierung als funktionale Programme. Man betrachte zum Beispiel Mengen über ganzen Zahlen als abstrakten Datentyp:

```
abstype IntSet =
  empty  : IntSet
  insert : (Int, IntSet) → IntSet
  member : (Int, IntSet) → Bool

axioms
  member (x, empty) = false
  member (x, insert (x, s)) = true
  member (x, insert (y, s)) = member (x, s)        if x ≠ y
```

Hier ist eine Implementierung dieses abstrakten Datentyps:


```
data IntSet = Empty | Insert (Int, IntSet)

empty  : IntSet
insert : (Int, IntSet) → IntSet
member : (Int, IntSet) → Bool

empty  = Empty
insert = Insert

member (x, Empty) = False
member (x, Insert (y, s)) = if x == y then True
                            else member (x, s)
```

An diesem Beispiel wird folgendes Implementierungsprinzip deutlich:

Nehmen wir an, die Operationen eines abstrakten Datentyps lassen sich in Generatoren G und abhängige Operationen A einteilen, so daß alle Axiome von der Form

$$A(G\ldots,\ldots,G\ldots) \quad = \quad \ldots$$

sind. In diesem Fall ensprechen die Generatoren G den Konstruktoren eines algebraischen Datentyps. Alle abhängigen Operationen A sind Funktionen, und jedes Axiom ist eine definierende Gleichung einer Funktion.

Dieses Prinzip gibt uns auf einfachste Weise eine ausführbare Implementierung für einen abstrakten Datentyp, wenn dieser die G-A-Eigenschaft hat. Natürlich ist dies nicht die einzig mögliche Implementierung — eine Effizienzsteigerung kann oft durch Wechsel der Datenstruktur erreicht werden. Beispiel:

```
type IntSet = IntTree

empty = E

insert (x, E) = B x E E
insert (x, B y l r) =
  if x < y then B y (insert (x, l)) r
  else if y < x then B y l (insert (x, r))
  else B y l r

member (x, E) = False
member (x, B y l r) =
  if x < y then member (x, l)
```

```
else if y < x then member (x, r)
else True
```

Ein Nachweis der Korrektheit dieser Implementierung geschieht mit struktureller Induktion. Im einzelnen:

```
member (x, empty)
  =  member (x, E)
  =  False

member (x, insert (x, E))
  =  member (x, B x E E)
  =  True
```

Im Fall member (x, insert (x, B y l r)) unterscheiden wir, ob $x < y$, $x = y$, oder $x > y$. Wenn $x < y$, gilt:

```
member (x, insert (x, B y l r))
  =  member (x, B y (insert (x, l)) r)
  =  member (x, insert (x, l))
  =  True               (nach der Induktionshypothese)
```

Die anderen Fälle werden analog bewiesen.

Aufgabe 6.1 Vervollständigen Sie den obenstehenden Beweis.

6.3 Polymorphie

Die Programmierung von IntTree in den letzten Abschnitten könnte ohne weiteres auf Bäume beliebiger Elemente erweitert werden. Dazu muß aber der Typ IntTree zu einem *polymorphen*, d. H. "vielförmigen" Typ verallgemeinert werden. Dies kann durch Parametrisierung der Typdefinition ausgedrückt werden.

```
data Tree a = E
              B a (Tree a) (Tree a)
```

Die Typvariable a kann zu jedem beliebigen Typ instanziiert werden. Dies muß aber überall in gleicher Weise geschehen. Zum Beispiel ist B 1 (B 2 E E) E ein Wert des Typs Tree Int, aber B 1 (B True E E) E paßt nicht in die Tree-Definition, da verschiedene Elementtypen vorkommen.

Die Typisierung von flip ist nun:

```
flip : Tree a → Tree a.
```

Dies sagt aus, daß flip einen Baum mit einem beliebigen Elementtyp a als Argument erwartet und einen Baum vom gleichen Elementyp zurückgibt. An den definierenden Gleichungen von flip ändert sich nichts.

Diese Art von Polymorphie heißt auch ***parametrische Polymorphie*** oder ***Generizität***. Eine andere Art von Polymorphie ist die ***Untertyp-Polymorphie***, die dadurch entsteht, daß statt eines Ausdrucks eines gegebenen Typs T auch ein Ausdruck eines beliebigen Untertyps von T verwendet werden kann. Kennzeichnend für die parametrische Polymorphie ist die Möglichkeit, Typen über gemeinsame Typvariablen zu koppeln.

Beispiel 6.1 Eine Verallgemeinerung der Implementierung von IntSet auf Mengen mit beliebigem Elementtyp.

```
type Set a = Tree a

  empty   : Set a
  insert  : (a, Set a) → Set a
  member : (a, Set a) → Bool

  empty = E

  insert (x, E) = B x E E
  insert (x, B y l r) =
    if x < y then B y (insert (x, l)) r
    else if y < x then B y l (insert (x, r))
    else B y l r

  member (x, E) = False
  member (x, B y l r) =
    if x < y then member (x, l)
    else if y < x then member (x, r)
    else True
```

Außer der Typdefinition und den Typisierungen hat sich nichts geändert.

6.4 Funktionen höherer Ordnung

Der Code in Beispiel 6.1 wirft möglicherweise ein Problem auf: Welche Vergleichsoperation (<) wird aufgerufen, wenn der Elementtyp statisch nicht bekannt ist? Viele funktionale Programmiersprachen (wenn auch nicht Haskell) verbieten deshalb auf polymorphen Typen die Benutzung eines überladenen Operators wie Z. B. (<).

Das Problem kann gelöst werden, indem den Funktionen insert und member die zu verwendende Vergleichsoperation explizit übergeben wird. Also:

```
type Set a = Tree a

  empty  : Set a
  insert : ((a, a) → Bool) → (a, Set a) → Set a
  member : ((a, a) → Bool) → (a, Set a) → Bool

  empty = E

  insert less (x, E) = B x E E
  insert less (x, B y l r) =
    if less (x, y) then B y (insert (x, l)) r
    else if less (y, x) then B y l (insert (x, r))
    else B y l r

  member less (x, E) = False
  member less (x, B y l r) =
    if less(x, y) then member (x, l)
    else if less(y, x) then member (x, r)
    else True
```

Um Schreibarbeit zu vermeiden, können wir insert und member lokal mit einer Vergleichsoperation spezialisieren:

```
intless        : (Int, Int) → Bool
intless (x, y) = x < y

intInsert      = insert intLess
intMember      = member intLess
```

Eine Funktion, die wie member eine andere Funktion als Argument übernimmt, oder die wie intMember eine andere Funktion als Resultat zurückgibt, heißt *Funktion höherer Ordnung*. Solche Funktionen werden im FP oft verwendet. Sie tragen einen Großteil zur Mächtigkeit dieses Programmierstils bei.

Ein Spezialfall ist die sogenannte "Curry"-Schreibweise, in der Funktionsargumente nicht mehr durch Klammern zusammengefaßt werden. Man betrachte zum Beispiel die folgende Funktion:

```
comp : (b → c) → (a → b) → (a → c)
comp f g x = f (g x)
```

Dies definiert eine *einstellige* Funktion mit formalem Argument f. Die Funktion gibt eine weitere Funktion als Resultat zurück. Die Resultatsfunktion ist wieder einstellig mit formalem Argument g. Sie gibt wiederum eine Resultatsfunktion zurück, die schließlich ein x übernimmt und nach f (g x) abbildet. Das heißt statt einer dreistelligen Funktion, haben wir drei einstellige Funktionen. Eine äquivalente Schreibweise mit lokalen Definitionen ist:

```
comp f = let comp1 g = let comp2 x = f (g x)
                       in comp2
         in comp1
```

Die "Curry"-Schreibweise erlaubt die einfache Spezialisierung einer Funktion mit einem Teil ihrer Argumente, wie im Fall von intInsert, intMember geschehen. Sie hat oft auch den Vorteil, daß Klammern gespart werden, weil wir folgende Konventionen aufstellen:

```
A → B → C = A → (B → C)
f a b     = (f a) b
```

Das heißt Funktionstypen klammern nach rechts und Funktionsanwendung klammert nach links.

Zum Schluß dieses Abschnitts führen wir noch zwei notationelle Vereinfachungen für Funktionen ein.

Erstens erlaubt es die λ-Notation, namenlose Funktionen zu schreiben. So bezeichnet zum Beispiel

$$\lambda x.x * x$$

die Quadratfunktion, ohne daß wir diese Funktion mit einen Namen versehen müßten.

Zweitens lassen wir auch Binäroperatoren in Infix-Notation zu. Zum Beispiel kann der Benutzer comp so als Infix-Operator definieren:

```
infix ∘
f ∘ g = λx. f (g x)
```

Aufgabe 6.2 Was ist der Typ folgender Funktionen?

```
apply x f         = f x
curry f x y       = f (x, y)
uncurry f (x, y)  = f x y
```

6.5 Listen

Der Datentyp "Liste" kommt so oft in funktionalen Programmen vor, daß er fast immer durch spezielle Syntax ausgedrückt wird. Eine Liste von Elementen a_1, ..., a_n wird $[a_1, ..., a_n]$ geschrieben. [a] bezeichnet den Typ aller Listen von Elementen vom Typ a. Die Infixoperation (:) fügt ein Element vorne an eine Liste an.

$$a : [a_1, ..., a_n] = [a, a_1, ..., a_n]$$

(:) ist rechtsassoziativ — a : b : cs ist also äquivalent zu a : (b : cs). Listen können als algebraischer Datentyp aufgefaßt werden:

```
type [a] = [] | a : [a]
```

Die Form $[a_1, ..., a_n]$ ist dann eine Abkürzung für

$$a_1 : a_2 : ... : a_n : []$$

Hier sind einige oft benutzte Operationen auf Listen:


```
head            : [a] → a
head []         = error "head"
head (x:xs)     = x

tail            : [a] → [a]
tail []         = error "tail"
tail (x:xs)     = xs

take            : Int → [a] → [a]
take 0 xs       = []
take n []       = []
take n (x:xs)   = x : take (n−1) xs

drop            : Int → [a] → [a]
drop 0 xs       = xs
drop n []       = []
drop n (x:xs)   = drop (n−1) xs
```

head und tail geben das erste Element und den Rest einer Liste zurück. take und drop geben die ersten n Elemente einer Liste bzw. den Rest ohne die ersten n Elemente zurück. Die error-Funktion bricht bei Aufruf das Programm ab — sie entspricht also in etwa der exit-Funktion in C-Bibliotheken.

Andere Operationen lassen sich aus den oben definierten synthetisieren, z. B. :

```
nth       : Int → [a] → a
nth n xs = head (drop n xs)
```

Die direkte Definition von nth kann wie folgt hergeleitet werden.

```
nth n [] =  head (drop n [])     nach Definition von nth
         =  head []              nach Definition von drop
         =  Error "head"         nach Definition von head

nth 0 (x:xs) =  head (drop 0 (x:xs)    nach Definition von nth
             =  head (x:xs)            nach Definition von drop
             =  x                      nach Definition von head
```

Und wenn n ≠ 0:

```
nth n (x:xs) =  head (drop n (x:xs)    nach Definition von nth
             =  head (drop (n−1) xs    nach Definition von drop
             =  nth (n−1) xs           nach Definition von nth
```

Also zusammengefaßt:

```
nth n []      = error "head"
nth 0 (x:xs)  = x
nth n (x:xs)  = nth (n−1) xs
```

Die Technik der Ableitung von nth hat lediglich die definierenden Gleichungen von Funktionen angewandt, und zwar in beide Richtungen. Sie heißt *fold-unfold*-Methode.

Aufgabe 6.3 Definieren Sie eine Funktion empty, die angibt, ob eine Liste leer ist oder nicht.

Die Konkatenation zweier Listen xs und ys wird xs ++ ys geschrieben. Der Operator (++) ist wie folgt definiert.

```
(++)          : [a] → [a] → [a]
[] ++ ys      = ys
(x:xs) ++ ys  = x : (xs ++ ys)
```

Die n-stellige Verallgemeinerung von (++) heißt concat. concat nimmt eine Liste von Listen $[xs_1,\dots,xs_n]$, und konkateniert alle Teillisten zu xs_1 ++ ... ++ xs_n. Der Typ von concat ist also wie folgt definiert:

```
concat : [[a]] → [a].
```

Aufgabe 6.4 Geben Sie eine Definition von concat an.

Eine weitere Kurzschreibweise für Listen drückt arithmetische Intervalle aus; Z. B. kann der Ausdruck [1,2,3,4,5] zu [1..5] verkürzt werden.

Aufgabe 6.5 Definieren Sie eine Funktion range, so daß range x y = [x..y].

6.5.1 Listenfunktionen höherer Ordnung

Listenoperationen können flexibler gestaltet werden, wenn Funktionen als Argumente übergeben werden. Hier ist eine Funktion, die alle Elemente einer Liste auswählt, die eine bestimmte Bedingung p erfüllen.


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

Hier ist eine Funktion, die das längste Präfix einer Argumentliste zurückgibt, dessen Elemente alle eine gegebene Bedingung erfüllen.

```
takewhile          : (a → Bool) → [a] → [a]
takewhile p []     = []
takewhile p (x:xs) = if p x then x : takewhile p xs
                     else []
```

Aufgabe 6.6 Definieren Sie eine Funktion dropwhile, die alle Elemente vom Anfang einer Liste wegnimmt, die eine bestimmte Bedingung erfüllen.

Hier ist eine Funktion, die eine Argument-Funktion auf jedes Element einer Argument-Liste anwendet.

```
map          : (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Oft wird auch ein *Reduktionsoperator* benötigt, der eine gegebene Binäroperation $(\oplus)$ zwischen alle Elemente einer Liste einfügt. Dabei sind zwei Möglichkeiten denkbar, je nachdem, ob nach links oder nach rechts geklammert wird. Wir definieren also für n >= 0:

$$\mathsf{foldl1}\ (\oplus)\ [x_1,\dots,x_n] = (\dots(x_1 \oplus x_2) \oplus \dots \oplus x_n)$$

$$\mathsf{foldr1}\ (\oplus)\ [x_1,\dots,x_n] = (x_1 \oplus \dots \oplus (x_{n-1} \oplus x_n)\dots)$$

Es ist oft sinnvoll, noch ein Null-Element hinzuzufügen, unter anderem, um auch die leere Liste behandeln zu können. Für beliebiges n soll also gelten:

$$\mathsf{foldl}\ (\oplus)\ z\ [x_1,\dots,x_n] = (\dots(z \oplus x_1) \oplus \dots \oplus x_n)$$
$$\mathsf{foldr}\ (\oplus)\ z\ [x_1,\dots,x_n] = (x_1 \oplus \dots \oplus (x_n \oplus z)\dots)$$

Hier sind die Definitionen von foldl und foldr:

```
foldl             : (a → b → a) → a → [b] → a
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs

foldr             : (b → a → a) → a → [b] → a
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

Aufgabe 6.7 Geben Sie Typen und Definitionen für die Funktionen foldl1 und foldr1 an.

Mit den hier definierten Listenfunktionen höherer Ordnung lassen sich viele gängige Listenoperationen konzis und ohne explizite Rekursion ausdrücken. Hier sind einige Beispiele:

Summe und Produkt der Elemente einer Liste:

```
sum     = foldl (+) 0
product = foldl (*) 1
```

Die Länge einer Liste:

```
length xs = sum (map (λx.1) xs)
```

Die Elemente einer Liste in umgekehrter Reihenfolge:

```
reverse xs = let snoc ys y = y : ys
             in  foldl snoc [] xs
```

Die xs-Argumente der letzten beiden Funktionen sind eigentlich überflüssig — genausogut ließen sich die Funktionen so definieren:

```
length  = sum ∘ (map (const 1))
snoc    = flip (:)
reverse = foldl snoc []
```

Hier haben wir folgende primitive Funktionen vorausgesetzt:

```
flip f x y  = f y x
const x y = x
```

Ein Stil, der auf explizite Funktionsargumente verzichtet und statt dessen Funktionen aus einer vorgegebenen Bibliothek von Basisfunktionen zusammensetzt, heißt *Kombinator-Stil*. Manche Sprachen verwenden ausschließlich diesen Stil; Beispiele sind APL [9] oder FP [2].

6.5.2 Listenausdrücke

Manche funktionalen Sprachen erlauben es, Listen mit einer Schreibweise zu definieren, die an die übliche Mengenschreibweise angelehnt ist. Hier ist ein Beispiel:

```
[ x ∗ x | x ← [0..9], odd x]
```

Dieser Ausdruck ergibt die Liste [1, 9, 25, 49, 81]. Ein solcher Listenausdruck setzt sich zusammen aus einem definierenden Ausdruck links vom Trennungsstrich | und einer Reihe von *Generatoren* und *Filtern* rechts davon. Ein Generator x ← e gibt einen Wertebereich e an, über den ein Bezeichner x variiert. Jeder Listenausdruck muß mindestens einen Generator haben. Sind mehrere Generatoren vorhanden, so variieren später definierte Bezeichner schneller als früher definierte; Z. B. ergibt der Ausdruck

```
[(x, y) | x ← [0..1], y ← ['a', 'b']]
```

die Liste

```
[(0, 'a'), (0, 'b'), (1, 'a'), (1, 'b')].
```

Ein Filter p wählt aus den vorher definierten Listenelementen nur solche aus, die das Prädikat p erfüllen.

Beispiel 6.2 Hier ist eine Funktion, die einen azyklischen Graphen, der durch eine Knotenliste und eine Kantenliste gegeben ist, topologisch sortiert.

```
type Node  = Int
type Edge  = (Node, Node)
type Graph = ([Node], [Edge])

topsort : Graph → [Node]
topsort (nodes, edges) =
  let preds x     = [y | (y, x') ← edges, x == x']
      sources     = [x | x ← nodes, empty (preds x)]
      elem x xs   = not (empty [x' | x' ← xs, x == x'])
      restnodes   = [x | x ← nodes, not (elem x sources)]
      restedges   = [(x, y) | (x, y) ← edges, not (elem x sources)]
  in  sources ++ topsort (restnodes, restedges)
```

Listausdrücke vereinfachen oft die Schreibweise, sind aber nicht unbedingt notwendig. Zum Beispiel könnte der eingangs definierte Ausdruck

```
[ x * x | x ← [0..9], odd x]
```

auch so definiert werden:

```
let square x = x * x
in  map square (filter odd [0..9]).
```

Die Liste von Paaren

```
[(x, y) | x ← [0..1], y ← ['a', 'b']]
```

ließe sich auch so ausdrucken:

```
let pair x y  = (x, y)
    sublist x = map (pair x) ['a', 'b']
in  concat (map sublist [0..1]).
```

Man kann zeigen, daß sich jeder Listenausdruck durch die Funktionen map, filter und concat darstellen läßt. Das Übersetzungsschema ist wie folgt.

- [e | x ← g] wird übersetzt zu map (λx.e) g.
- [e | x ← g, p, ...] wird übersetzt zu [e | x ← filter (λx.p) g,...]
- [e | x ← g, y ← g', ...] wird übersetzt zu concat [[e | y ← g', ...] | x ← g].

Umgekehrt können map, filter und concat auch durch Listenausdrücke dargestellt werden:

```
map f xs    = [f x | x ← xs]
filter p xs = [x | x ← xs, p]
concat xss  = [x | xs ← xss, x ← xs].
```

6.6 Auswertungsreihenfolge

Bei der Auswertung funktionaler Programme sind zwei Strategien möglich.

Die *strikte* Auswertung wertet alle Argumente aus, bevor sie an eine Funktion übergeben werden. Die *faule* Auswertung übergibt Funktions-Argumente unausgewertet. Ein Argument wird erst ausgewertet, wenn sein Wert für den Fortgang der Berechnung benötigt wird. Dasselbe gilt für Datenstrukturen: Eine Liste — bzw. der Rest einer Liste — wird nur dann ausgewertet, wenn ihre Elemente benötigt werden.

Beispiel 6.3 Betrachte eine Disjunktion über einer Liste von Wahrheitswerten:

```
ex        : [Bool] → Bool
ex []     = false
ex (x:xs) = x || ex xs
```

ex läßt sich kürzer so ausdrücken:

```
ex = foldr (||) false
```

Bei fauler Auswertung ist die Abarbeitung beider Programme im Prinzip dieselbe: Die Listenelemente werden ausgewertet und mit (||) verknüpft, bis ein true-Element gefunden wird oder bis das Ende der Liste erreicht ist. Bei strikter Auswertung gibt es dagegen einen Unterschied. Das rekursive Programm wird abgearbeitet wie vorher. Das Programm mit foldr wird dagegen so ausgewertet, daß immer alle Verknüpfungen durchgeführt werden.

Aufgabe 6.8 Welcher der Faltungsoperatoren foldl, foldr ist effizienter, wenn die Binäroperation assoziativ ist und die Programmiersprache strikt ist? Ändert sich das, wenn die Programmiersprache nicht strikt ist?

Kapitel 7

Kalküle für Funktionen

7.1 Der Lambda-Kalkül

Der Lambda-Kalkül ist eine einfache operationelle Semantik für funktionale Programme. Er wurde von Alonzo Church in den 30er und 40er Jahren als Theorie für berechenbare Funktionen entworfen, ursprünglich ohne dabei eine maschinelle Ausführung im Auge zu haben. Der Lambda-Kalkül ist genauso mächtig wie Turing-Maschinen, d. H. jede Turing-Maschine kann als Funktion im Lambda-Kalkül ausgedrückt werden und umgekehrt. Die *Hypothese von Church* besagt, daß jeder berechenbare Algorithmus durch eine lambda-definierbare Funktion (oder, äquivalenterweise, durch eine Turing-Maschine) ausgedrückt werden kann.

Der *reine* Lambda-Kalkül beschreibt ausschließlich Funktionen und Funktionsanwendungen, und zwar mit drei Termformen.

Variablen	$x, y, z \in \mathcal{V}$	
Terme	d, e, f	$::= \quad x \mid \lambda x.e \mid d\,e$

Ein Term ist entweder eine *Variable* x, eine *Lambda- Abstraktion* $\lambda x.e$, oder eine Funktionsanwendung $d\,e$. Informell stellt der Term $\lambda x.e$ eine Funktion dar, die jedem Argument x das Resultat der Berechnung von e zuordnet. Wir setzen ein unendlich großes Alphabet $\mathcal{V}$ von Variablen voraus. Wir verwenden im folgenden die Buchstaben x, y, z für Variablen und die Buchstaben d, e, f für Terme.

Funktionsanwendung ist linksassoziativ. Der Geltungsbereich einer λ-Abstraktion erstreckt sich so weit nach rechts wie möglich, d. H. bis zum

Ende des Terms oder bis zur schließenden Klammer. Beispiel:

$$\lambda f.\lambda x.fex \quad \equiv \quad (\lambda f.(\lambda x.((fe)x))).$$

Die Auswertung von Lambda-Ausdrücken erfolgt nach der β-Regel. Diese besagt, daß eine Funktionsanwendung $(\lambda x.d)\,e$ vereinfacht werden kann, indem das Funktionsargument e für den formalen Parameter x der Funktion $\lambda x.d$ eingesetzt wird.

$$\beta: \qquad (\lambda x.d)\,e \quad \rightarrow \quad [e/x]d.$$

Die genaue Bedeutung des Substitutionsoperators $[d/x]$ wird weiter unten erklärt.

Beispiel 7.1

$$\begin{array}{lcl} (\lambda x.x)(\lambda y.y) & \rightarrow & \lambda y.y \\ \\ (\lambda f.\lambda x.f\ (f\ x))(\lambda y.y)z & \rightarrow & (\lambda y.y)((\lambda y.y)z) \\ & \rightarrow & (\lambda y.y)z \\ & \rightarrow & z \end{array}$$

Freie und gebundene Variablen; Variablen-Umbenennung. Die freien Variablen $\mathrm{fv}(e)$ eines Terms e sind wie folgt definiert:

$$\mathrm{fv}(x) \ = \ \{x\} \qquad \mathrm{fv}(\lambda x.e) \ = \ \mathrm{fv}(e)\backslash\{x\} \qquad \mathrm{fv}(f\,e) \ = \ \mathrm{fv}(f) \cup \mathrm{fv}(e).$$

Alle anderen Variablen, die in einem Term vorkommen, heißen *gebunden*. Ein Term heißt *abgeschlossen*, wenn er keine freien Variablen enthält.

Die Namen gebundener Variablen sind nicht signifikant, d. H. wir betrachten zwei Terme d und e, die durch konsistente Umbenennung von gebundenen Variablen auseinander hervorgehen, als identisch und schreiben $d \equiv e$. Formal wird das durch folgende Umbenennungsregel ausgedrückt:

$$\alpha: \qquad \lambda x.e \quad \equiv \quad \lambda y.[y/x]e \qquad (y \notin \mathrm{fv}(e))$$

Substitution betrifft nur die freien Variablen eines Terms:

$$\begin{array}{lcll} [d/x]x & = & d & \\ [d/x]y & = & y & (x \neq y) \\ [d/x]\lambda y.e & = & \lambda y.[d/x]e & (x \neq y, y \notin \mathrm{fv}(d)) \\ [d/x](f\,e) & = & ([d/x]f)\,([d/x]e) & \end{array}$$

Außerdem ist darauf zu achten, daß freie Variablen des substituierten Ausdrucks nicht gebunden werden. Z. B. ergibt $[y/x]\lambda y.x$ nicht etwa $\lambda y.y$, sondern ist erst nach Umbenennung definiert:

$$[y/x]\lambda y.x \ \equiv \ [y/x]\lambda z.x \ \equiv \ \lambda z.y.$$

Im folgenden nehmen wir immer an, daß Umbenennungen automatisch soweit vorgenommen werden, daß alle Substitutionen definiert sind.

Beispiel 7.2 Definiere

$$\begin{array}{l} S \stackrel{\mathrm{def}}{\equiv} \lambda f.\lambda g.\lambda x.fx(gx) \\ K \stackrel{\mathrm{def}}{\equiv} \lambda x.\lambda y.x \end{array}$$

Dann können wir den Term SKK wie folgt reduzieren.

$$\begin{array}{rcl} SKK & \stackrel{\mathrm{def}}{\equiv} & (\lambda f.\lambda g.\lambda x.fx(gx))KK \\ & \rightarrow & (\lambda g.\lambda x.Kx(gx))K \\ & \rightarrow & \lambda x.Kx(Kx) \\ & \stackrel{\mathrm{def}}{\equiv} & \lambda x.(\lambda y.\lambda z.y)x(Kx) \\ & \rightarrow & \lambda x.(\lambda z.x)(Kx) \\ & \rightarrow & \lambda x.x \end{array}$$

Man bemerke daß wir eine umbenannte Version von K verwendet haben, um einem Namenskonflikt aus dem Weg zu gehen.

Wir schreiben $\twoheadrightarrow$ für Reduktion in beliebig vielen Schritten:

$$e \twoheadrightarrow e' \qquad \text{genau dann, wenn} \qquad \exists n \geq 0.e \equiv e_0 \rightarrow \ldots \rightarrow e_n \equiv e'$$

Terme, die sich nicht weiter reduzieren lassen, heißen *Normalformen*. Es ist ein beliebter Zeitvertreib, λ-Terme auf ihre Normalformen zu reduzieren. Der folgende Satz impliziert, daß Normalformen eindeutig sind.

Satz 7.3 (Church-Rosser) Reduktion im Lambda-Kalkül ist *konfluent*: Falls $e \twoheadrightarrow e_1$ und $e \twoheadrightarrow e_2$, dann existiert ein Term e_3, so daß $e_1 \twoheadrightarrow e_3$ und $e_2 \twoheadrightarrow e_3$.

Korollar 7.4 Jeder Term hat höchstens eine Normalform.

Beweis: Nimm an, ein Term e habe zwei Normalformen e_1 und e_2. Nach Satz 7.3 gibt es dann einen Term e_3 so daß $e_1 \twoheadrightarrow e_3$ und $e_2 \twoheadrightarrow e_3$. Da Normalformen nicht weiter reduzierbar sind, muß gelten $e_1 \equiv e_3 \equiv e_2$. $\Box$

Es gibt allerdings Terme, die keine Normalform haben. Ein Beispiel ist

$$\begin{array}{rcl} \Omega & \stackrel{\text{def}}{=} & (\lambda x.(xx))(\lambda x.(xx)) \\ & \rightarrow & (\lambda x.(xx))(\lambda x.(xx)) \\ & \rightarrow & (\lambda x.(xx))(\lambda x.(xx)) \\ & \rightarrow & \ldots \end{array}$$

Diese Beobachtung wirft nun die Frage nach einer geeigneten Auswertungsstrategie auf, die nichtterminierende Berechnungen möglichst vermeidet. Implementierungen von funktionalen Sprachen werten allerdings in der Regel nicht bis zu einer Normalform aus, sondern halten an, sobald der reduzierte Term eine Lambda-Abstraktion oder eine Variable ist. Solche Terme werden auch *schwache Normalformen* oder *Werte* genannt:

$$v \quad ::= \quad \lambda x.e \;\mid\; x.$$

Gibt es also eine *vollständige* Auswertungstrategie, die immer zu einer schwachen Normalform führt, wenn eine existiert? Man kann zeigen, daß die Strategie, die immer den Unterterm reduziert, der am weitesten links außen liegt, in diesem Sinne vollständig ist. Wir machen das präzise durch Einführung von *Auswertungskontexten*, die uns die Stelle des nächsten zu reduzierenden Terms angeben.

Definition. Ein λ-*Auswertungskontext* E ist ein Kontext in der Sprache

$$E ::= [\,] \mid E\,e$$

Standardreduktion $\underset{s}{\rightarrow}$ ist durch folgendes Ableitungssystem gegeben:

$$(\lambda x.d)\,e \underset{s}{\rightarrow} [d/x]e$$

$$\frac{d \underset{s}{\rightarrow} e}{E[d] \underset{s}{\rightarrow} E[e]}$$

Definition. Ein Term e *konvergiert*, in Symbolen $e \Downarrow$, wenn $e \twoheadrightarrow v$ für einen beliebigen Wert v.

Definition. Ein Term e *konvergiert* mittels Standardreduktion, in Symbolen $e \Downarrow_s$, wenn $e \underset{s}{\twoheadrightarrow} v$ für einen beliebigen Wert v.

Standardreduktion ist eine *vollständige* Auswertungsstrategie — wenn ein Term konvergiert, dann konvergiert er auch mittels Standardreduktion:

Satz 7.5 Für alle Terme e gilt: $e \Downarrow$ genau dann, wenn $e \Downarrow_s$.

Die "links-außen-Strategie" der Standardreduktion wird allerdings von Implementierungen in der Regel nicht eingesetzt, weil sie zu hohem Aufwand wegen wiederholter Berechnung von Argumenten führen kann. Beispiel:

$$\begin{array}{ll} \multicolumn{2}{l}{(\lambda f.\lambda x.f(fx))((\lambda x.x)(\lambda x.x))} \\ \rightarrow & \lambda x.(\lambda x.x)(\lambda x.x)((\lambda x.x)(\lambda x.x)x) \\ \rightarrow & \lambda x.(\lambda x.x)((\lambda x.x)(\lambda x.x)x) \\ \rightarrow & \lambda x.(\lambda x.x)((\lambda x.x)x) \\ \rightarrow & \lambda x.(\lambda x.x)x \\ \rightarrow & \lambda x.x \end{array}$$

Man bemerke, daß der Term $(\lambda x.x)(\lambda x.x)$ zweimal ausgewertet wird. Dies kann vermieden werden, wenn das Funktionsargument *vor* der Einsetzung in die Funktion ausgewertet wird. Anders ausgedrückt, wird immer der linkeste innerste reduzierbare Unterterm ausgewertet. In unserem Beispiel spart dies eine Reduktion:

$$\begin{array}{ll} \multicolumn{2}{l}{(\lambda f.\lambda x.f(fx))((\lambda x.x)(\lambda x.x))} \\ \rightarrow & (\lambda f.\lambda x.f(fx))(\lambda x.x) \\ \rightarrow & \lambda x.(\lambda x.x)((\lambda x.x)x) \\ \rightarrow & \lambda x.(\lambda x.x)x \\ \rightarrow & \lambda x.x \end{array}$$

Die beiden Auswertungsstratgien werden *Namensaufruf* (für links-außen) und *Wertaufruf* (für links-innen) genannt. In der Praxis ist der Effizienzgewinn von Wertaufruf gegenüber Namensaufruf oft viel größer als an obigem Beispiel. Allerdings ist Wertaufruf keine vollständige Auswertungsstrategie.

Aufgabe 7.1 Nennen Sie einen Term, der sich mit Namensaufruf, aber nicht mit Wertaufruf zu einer schwachen Normalform reduzieren läßt.

Zusammengefaßt ist Namensaufruf zwar eine vollständige Strategie, aber in der Praxis bei weitem zu ineffizient, während Wertaufruf effizient, aber nicht vollständig ist. Es bieten sich zwei Wege an, um aus diesem Dilemma herauszukommen.

Zum einen kann die Namensaufrufstrategie verbessert werden, indem die wiederholte Neuberechnung von Argumenten vermieden wird. Dazu werden unausgewertete Argumentausdrücke in Speicherzellen geschrieben. Beim ersten Zugriff auf ein Argument wird der Argumentausdruck ausgewertet und danach die Speicherzelle des Arguments mit dem Resultatwert überschrieben. Spätere Zugriffe benutzen dann diesen Wert, anstatt das Argument erneut auszuwerten. Dazu ist es günstig, wenn Werte und Ausdrücke dasselbe Format haben. Diese Strategie wird *verzögerte* oder *faule* Auswertung genannt.

Aufgabe 7.2 Überlegen Sie, was ein günstiges Datenformat für Argumente bei der verzögerten Auswertungsstrategie ist.

Verzögerte Auswertung liefert dieselben Resultate wie Namensaufruf, ist aber deutlich effizienter. Theoretisch ist verzögerte Auswertung auch mindestens so effizient wie Wertaufruf, d. H. es werden nicht mehr Reduktionsschritte benötigt.

Aufgabe 7.3 Warum?

In der Praxis wiegen allerdings der erhöhte Aufwand zum Anlegen und Überschreiben einer Reduktion bei verzögerter Auswertung schwerer als die mögliche Vermeidung von Reduktionsschritten. Außerdem ist bei diesen Überlegungen nichts über den Speicherplatzbedarf einer Berechnung ausgesagt. Tatsächlich haben Berechnungen mit verzögerter Auswertung oft einen sehr hohen Speicherplatzbedarf, der noch dazu nur schwer abschätzbar ist.

Der andere Weg aus unserem Dilemma geht von der Wertaufrufstrategie aus, und ändert die formalen Grundlagen soweit, daß Wertaufruf zu einer vollständigen Auswertungsstrategie wird. Dies wird im nächsten Abschnitt behandelt.

7.2 Der Lambda-Kalkül mit Wertaufruf

Um der gängigen und effizienten Wertaufruf-Strategie eine formale Grundlage zu geben, ändern wir den Lambda-Kalkül ein wenig ab. Wir führen eine Unterklasse von Termen ein, die Werte umfaßt:

$$\begin{array}{lrcl} \text{Werte} & v & ::= & x \mid \lambda x.e \\ \text{Terme} & e & ::= & v \mid e_1\, e_2 \end{array}$$

Ein Wert ist also eine Variable oder eine Lambda-Abstraktion. Die Reduktionsregel wird dann so modifiziert, daß nur Werte übergeben werden können.

$$\beta_v \qquad (\lambda x.d)\, v \quad \rightarrow_v \quad [v/x]d.$$

Der resultierende Kalkül heißt Lambda-Kalkül mit Wertaufruf, oder kurz λ_v. λ_v hat weitgehend analoge Eigenschaften zum klassischen Lambda-Kalkül.

Satz 7.6 λ_v ist konfluent.

Definition. Ein λ_v-Auswertungskontext E_v ist ein Satz der Sprache

$$E_v \quad = \quad [\,] \mid E_v\, e \mid v\, E_v$$

Standardreduktion $\underset{s}{\rightarrow}_v$ ist durch folgendes Ableitungssystem gegeben:

$$(\lambda x.d)\, e \underset{s}{\rightarrow}_v [d/x]e$$

$$\frac{d \underset{s}{\rightarrow}_v e}{E_v[d] \underset{s}{\rightarrow}_v E_v[e]}$$

Satz 7.7 $\underset{s}{\rightarrow}_v$ ist eine vollständige Auswertungsstrategie für Wertaufruf: Wenn $\exists v.e \twoheadrightarrow_v v$, dann $\exists v.e \underset{s}{\twoheadrightarrow}_v v$.

7.3 Church-Kodierung

Der vorletzte Abschnitt hatte ein reines Lambda-Kalkül behandelt, in dem lediglich anonyme Funktionen ausgedrückt werden können. Wir wissen bereits, daß dieses Kalkül Turing-mächtig ist, also jede berechenbare Funktion in ihm ausgedrückt werden kann. In diesem Abschnitt werden wir Konstrukte einer realistischen funktionalen Sprache im Lambda-Kalkül ausdrücken. Die behandelten Konstrukte sind beliebige algebraische Datentypen und Operationen auf ihnen sowie lokale, möglicherweise rekursive Definitionen. Dies entspricht ziemlich genau dem Unfang der im letzten Kapitel behandelten Programmiersprache. Interessanterweise ist die Kodierung aller dieser Konstrukte recht einfach und natürlich.

Zunächst zu Datentypen. Wir beginnen mit dem konkreten Beispiel des Typs

$$\mathsf{data\ bool} = \mathsf{true} \mid \mathsf{false}.$$

Wie sollen true und false als Lambda-Ausdrücke geschrieben werden? Aufschluß darüber bietet die Beobachtung, welche Operationen auf dem Typ bool ausgeführt werden. Alle solche Operationen können durch bedingte Ausdrücke ausgedrückt werden, z. B. :

$$\begin{array}{lcl}\mathsf{not}\ x & = & \mathsf{if}\ x\ \mathsf{then\ false\ else\ true}, \\ x\ \mathsf{or}\ y & = & \mathsf{if}\ x\ \mathsf{then\ true\ else}\ y, \\ x\ \mathsf{and}\ y & = & \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else\ false}.\end{array}$$

Die Idee der Church-Kodierung ist nun, den Booleschen Wert x genauso wie die zweistellige Funktion if x then_ else _ zu kodieren. Das heißt:

$$\begin{array}{lcl}\mathsf{true} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.x, \\ \mathsf{false} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.y, \\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \stackrel{\text{def}}{=} & e_1\, e_2\, e_3.\end{array}$$

Beispiel 7.8

$$\begin{array}{lcl}\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \stackrel{\text{def}}{=} & (\lambda x.\lambda y.x)\, e_1\, e_2 \\ & \longrightarrow & (\lambda y.e_1)\, e_2 \\ & \longrightarrow & e_1\end{array}$$

Dieses Prinzip läßt sich auf beliebige algebraische Datentypen verallgemeinern, nur daß für diese der universelle Selektor der case- Ausdruck ist. Wieder ist das Prinzip, daß x genauso wie case x of... kodiert wird. Hier ist das Beispiel des Listentyps,

```
data List a = Nil | Cons a (List a)
```

mit Selektor

case xs of Nil $\Rightarrow e_1$ | Cons x xs' $\Rightarrow e_2$.

Im Vergleich zu der Kodierung von Booleschen Werten kommt hinzu, daß das Muster `Cons x xs'` Variablen `x` und `xs'` enthält, die im Ausdruck `e2` verwendet werden können. Dies wird in der Church-Kodierung so ausgedrückt, daß `e2` eine zweistellige Funktion ist, der die Werte von `x` und `xs'` übergeben werden. Also:

$$\begin{array}{lcl}\mathsf{nil} & \stackrel{\text{def}}{=} & \lambda n.\lambda c.n \\ \mathsf{cons}\ e_1\ e_2 & \stackrel{\text{def}}{=} & \lambda n.\lambda c.c\, e_1\, e_2\end{array}$$

Die Funktion


```
car xs = case xs of
           Nil        ⇒ Nil
           Cons x xs' ⇒ x
```

wird kodiert als:

$$\lambda xs.xs\ \mathsf{nil}\ (\lambda x.\lambda xs'.x)$$

Dann ist

$$\begin{array}{ll} & \mathsf{car(cons\ true\ nil)} \\ \stackrel{\text{def}}{=} & (\lambda xs.xs\ \mathsf{nil}\ (\lambda x.\lambda xs'.x))\ (\lambda n.\lambda c.c\ \mathsf{true\ nil}) \\ \longrightarrow & (\lambda n.\lambda c.c\ \mathsf{true\ nil})\ \mathsf{nil}\ (\lambda x.\lambda xs'.x) \\ \twoheadrightarrow & (\lambda x.\lambda xs'.x)\ \mathsf{true\ nil} \\ \longrightarrow & \mathsf{true}.\end{array}$$

Diese Beispiele führen nun zum Kodierungsprinzip für allgemeine Datentypen.

Sei D ein algebraischer Datentyp mit Konstruktoren $K_1, \ldots, K_m$ der Stelligkeiten $n_1, \ldots, n_m$. Dann wird ein Konstruktor-Ausdruck

$$K_i\, x_1, \ldots, x_{n_i}$$

kodiert als

$$\lambda k_1 \ldots \lambda k_m.k_i\, x_1, \ldots, x_{n_i}.$$

Der Selektor-Ausdruck

$$\begin{array}{llcl}\mathsf{case}\ e\ \mathsf{of} & K_1\, x_1, \ldots, x_{n_1} & \Rightarrow & e_1 \\ | & \ldots & & \\ | & K_m\, x_1, \ldots, x_{n_m} & \Rightarrow & e_m\end{array}$$

wird kodiert als

$$\begin{array}{l} e\ (\lambda x_1 \ldots \lambda x_{n_1}.e_1) \\ \quad \ldots \\ \quad (\lambda x_1 \ldots \lambda x_{n_m}.e_m).\end{array}$$

Wir kommen nun zur Kodierung von lokalen Definitionen let $x = d$ in e. Wenn die Definition nicht rekursiv ist, d. H. wenn x in d nicht vorkommt, ist die Kodierung einfach:

$$\mathsf{let}\ x = d\ \mathsf{in}\ e \stackrel{\text{def}}{=} (\lambda x.e)\, d \qquad (x \notin \mathrm{fv}(d))$$

Wenn x in d vorkommt, ist diese Kodierung nicht anwendbar, da sich die Variable x im Ausdruck $(\lambda x.e)\, d$ nicht im Sichtbarkeitsbereich von d befindet. Wir formen deshalb zunächst den Ausdruck in einen Ausdruck mit explizitem Fixpunktoperator um:

$$\mathsf{let}\ x = d\ \mathsf{in}\ e \stackrel{\text{def}}{=} \mathsf{let}\ x = \mathsf{fix}\ (\lambda x.d)\ \mathsf{in}\ e.$$

Dies reduziert also unser Problem auf die Kodierung des Fixpunktoperators fix. fix kann im reinen Lambda-Kalkül durch den Y-Kombinator dargestellt werden:

$$Y \stackrel{\text{def}}{=} \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)).$$

Dies kann wie folgt nachgepruft werden:

$$\begin{aligned} Y\,f &= (\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)) \\ &\rightarrow f\,((\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))) \\ &= f(Y\,f) \end{aligned} \quad .$$

Also ist $Y\,f$ Fixpunkt von f.

Kapitel 8

Typsysteme

Die Lambda-Sprache des letzten Kapitels war untypisiert. In diesem Kapitel werden wir studieren, was sich ändert, wenn Typen zum Lambda Kalkül hinzugenommen werden. Anders als in der untypisierten Welt gibt es dann keinen natürlichen Basiskalkül mehr, in dem alle Erweiterungen ausdrückbar sind. Wir werden deshalb mehrere Typsysteme behandeln, die teilweise aufeinander aufbauen:

- Der einfach getypte Lambda-Kalkül, F_1.
- Die Erweiterung von F_1 um explizite Typargumente, die durch den Lambda-Kalkül zweiter Ordnung, F_2, beschrieben wird.
- Die Erweiterung von F_1 um let-Ausdrücke und implizit polymorphe Funktionen, die im Hindley-Milner System, HM, behandelt wird.
- Die Erweiterung HM_O des Hindley-Milner Systems um überladene Funktionen und polymorphe Recordstrukturen

8.1 Der einfach getypte Lambda-Kalkül

Der einfach getypte Lambda-Kalkül unterscheidet sich vom ungetypten Lambda-Kalkül dadurch, daß wir nur noch *wohlgetypte* Terme betrachten, d. H. Terme, die im Ableitungssystem von Abbildung 11.1 in einer gegebenen Anfangstypothese Γ einen Typ haben. Dabei ist ein Typ τ ein Wort

Syntax

Typvariablen	α		
Typen	τ	$::=$	$\alpha \mid \tau \rightarrow \tau'$
Variablen	x		
Terme	e	$::=$	$x \mid \lambda x.e \mid e\ e'$

Typregeln

(TAUT) $\Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$

$$(\rightarrow\mathrm{I}) \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

$$(\rightarrow\mathrm{E}) \quad \frac{\Gamma \vdash e : \tau' \rightarrow \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

Abbildung 8.1: Der einfach getypte λ-Kalkül F_1

der folgenden Sprache:

$$\tau \quad ::= \quad \alpha \mid \tau \rightarrow \tau$$

Hier ist α eine *Typvariable*, für die ein beliebiger Typ eingesetzt werden kann. $\tau_1 \rightarrow \tau_2$ ist der Typ aller Funktionen, die Argumente vom Typ τ_1 in Resultate von Typ τ_2 abbilden. $\mathrm{tv}(\tau)$ bezeichnet die Menge aller Typvariablen in τ.

Die Klauseln des Ableitungssystems in Abbildung 11.1 haben die Form $\Gamma \vdash e : \tau$. Eine *Typumgebung* (oder *Typothese*) Γ ist eine endliche Menge von Bindungen $x : \tau$, wobei jedes x nur einmal in einer Typothese vorkommen kann.

Reduktion definiert sich wie vorher:

$$\beta : \qquad (\lambda x.d)\, e \rightarrow [e/x]d.$$

Wir zeigen nun einige wichtige Eigenschaften dieses Systems, aus denen folgt, daß Reduktion Typen unverändert läßt. Das erste Lemma sagt, daß Bindungen für Variablen, die in einem Ausdruck nicht vorkommen, redundant sind.

Lemma 8.1 Wenn $x \notin \mathrm{fv}(e)$, so gilt $\Gamma \vdash e : \tau$ genau dann, wenn $\Gamma, x : \tau' \vdash e : \tau$.

Aufgabe 8.1 Beweisen Sie Lemma 8.1.

Das zweite Lemma sagt, daß typkonforme Substitution Typen unverändert läßt.

Lemma 8.2 (Typ-Erhaltung von Substitution) Wenn $\Gamma, x : \tau' \vdash e : \tau$ und $\Gamma \vdash d : \tau'$, dann $\Gamma \vdash [d/x]e : \tau$.

Beweis: Durch Induktion über die Typableitung von e. Wir unterscheiden nach der letzten (untersten) Regel dieser Typableitung.

Case (TAUT): Dann ist der letzte Schritt von der Form

$$\Gamma, x : \tau' \vdash y : \tau \qquad (y : \tau \in \Gamma).$$

Wenn $x = y$, so gilt $\tau = \tau'$, da $\Gamma, x : \tau'$ nur einen Eintrag für x hat. Mit $\Gamma \vdash d : \tau'$ folgt, daß $\Gamma \vdash [d/x]x : \tau$. Wenn $x \neq y$, so gilt $[d/x]y = y$, also $\Gamma, x : \tau' \vdash [d/x]y : \tau$, also wegen Lemma 8.1 $\Gamma \vdash [d/x]y : \tau$.

Case $(\rightarrow\mathrm{I})$: Dann ist der letzte Schritt der Typableitung

$$\frac{\Gamma, x : \tau', y : \tau_1 \vdash e' : \tau_2}{\Gamma, x : \tau' \vdash \lambda y.e' : \tau_1 \rightarrow \tau_2}$$

für Typen τ_1, τ_2 so daß $\tau_1 \rightarrow \tau_2 = \tau$. Ohne Beschr nkung der Allgemeinheit sei $y \notin \mathrm{fv}(d)$ (wenn nicht, kann $\lambda y.e'$ mit der α Regel umbenannt werden). Wegen der Induktionshypothese gilt $\Gamma, y : \tau_1 \vdash [d/x]e' : \tau_1 \rightarrow \tau_2$. Also gilt nach Anwendung der Regel $(\rightarrow\mathrm{I})$, daß $\Gamma \vdash \lambda y.[d/x]e' : \tau_1 \rightarrow \tau_2$. Die Behauptung folgt dann, da $\lambda y.[d/x]e' \equiv [d/x]\lambda y.e'$, wegen $x \neq y$ und $y \notin \mathrm{fv}(d)$.

Case $(\rightarrow\mathrm{E})$: Dann ist der letzte Schritt der Typableitung

$$\frac{\Gamma, x : \tau' \vdash e_1 : \tau_2 \rightarrow \tau \qquad \Gamma, x : \tau' \vdash e_2 : \tau_2}{\Gamma, x : \tau' \vdash e_1\, e_2 : \tau}.$$

Wegen der Induktionshypothese gilt $\Gamma \vdash [d/x]e_1 : \tau_2 \rightarrow \tau$ und $\Gamma \vdash [d/x]e_2 : \tau_2$. Anwendung der Regel $(\rightarrow\mathrm{E})$ ergibt dann, daß $\Gamma \vdash ([d/x]e_1)([d/x]e_2) : \tau$. Daraus folgt mit $[d/x](e_1\, e_2) \equiv ([d/x]e_1)([d/x]e_2)$ die Behauptung. □

Der folgende Satz sagt, daß Typisierungen invariant unter Reduktion sind.

Satz 8.3 (Typ-Erhaltung von Reduktion) Wenn $\Gamma \vdash e : \tau$ und $e \to d$ dann gilt auch $\Gamma \vdash d : \tau$.

Beweis: Nimm an, daß $\Gamma \vdash (\lambda x.e)\, d : \tau$. Dann endet jede Ableitung dieses Typurteils in einem Baum der Form

$$\dfrac{\dfrac{\Gamma, x{:}\tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau}\,(\to\mathrm{I}) \qquad \Gamma \vdash d : \tau'}{\Gamma \vdash (\lambda x.e)\, d : \tau}\,(\to\mathrm{E})$$

Wenn wir Lemma 8.2 auf die Blätter dieses Baumes anwenden, ergibt sich, daß $\Gamma \vdash [d/x]e : \tau$. □

Eine weitere wichtige Eigenschaft des einfach getypten Lambda-Kalkül ist, daß jeder Term eine Normalform hat.

Satz 8.4 Jede Reduktionsfolge in F_1 endet in einer Normalform.

Diese Eigenschaft wird auch *starke Normalisierung* genannt. Ein Analogon zu Satz 8.4 gilt nicht im ungetypten Lambda-Kalkül, wo es Terme wie Ω gibt, die keine Normalform haben. Aus Satz 8.4 folgt direkt, daß F_1 nicht Turing-vollständig ist, also daß es berechenbare Funktionen gibt, die nicht in F_1 ausgedrückt werden können.

Aufgabe 8.2 Warum?

8.2 Typrekonstruktion

Im letzten Abschnitt haben wir ein Ableitungssystem für Typurteile in F_1 vorgestellt. In diesem Abschnitt untersuchen wir, ob die Typisierung in diesem Ableitungssystem entscheidbar ist. Dabei werden zwei Fragestellungen unterschieden:

1. Gegeben eine Typumgebung Γ, ein Term e, und ein Typ τ, kann entschieden werden, ob $\Gamma \vdash e : \tau$ gilt? Diese Frage wird auch *Typprüfungsproblem* genannt.

2. Gegeben eine Typumgebung Γ und ein Term e, kann entschieden werden, ob es einen Typ τ gibt, so daß $\Gamma \vdash e : \tau$? Diese Frage wird auch *Typrekonstruktionsproblem* genannt.

Nun können Terme in F_1 mehr als einen Typ haben. Beispiel:

$$\begin{array}{l} \lambda x.x : \alpha \to \alpha, \quad \text{und} \\ \lambda x.x : (\beta \to \beta) \to (\beta \to \beta). \end{array}$$

Bei der Typrekonstruktion stellt sich also die Frage, welcher dieser Typen zurückgegeben werden soll. In unserem obigen Beispiel würden wir den Typ $\alpha \to \alpha$ vorziehen, da dieser den Typ $(\beta \to \beta) \to (\beta \to \beta)$ als Instanz enthält. Diese Beziehung wird so formalisiert:

Definition. Ein Typ τ' ist eine *Instanz* eines Typs τ, in Symbolen $\tau' \preceq \tau$, wenn es eine Substitution S von Typvariablen nach Typen gibt, so daß $\tau' = S\tau$. In diesem Fall sagen wir auch, daß τ *allgemeiner* ist als τ'.

In F_1 hat jeder typisierbare Term e einen allgemeinsten Typ, der auch *prinzipieller Typ* von e genannt wird.

Satz 8.5 Entweder es gibt keinen Typ τ, so daß $\Gamma \vdash e : \tau$, oder es gibt einen Typ τ so daß gilt:

1. $\Gamma \vdash e : \tau$, und

2. Für alle τ': $\Gamma \vdash e : \tau' \;\Rightarrow \tau' \preceq \tau$.

Satz 8.5 ist eine Konsequenz der Korrektheit und Vollständigkeit des Typrekonstruktionsalgorithmus (Satz 8.8, Satz 8.9).

Das Typrekonstruktionsproblem für F_1 kann mit Satz 8.5 also folgendermaßen verschärft werden:

3. Gegeben eine Typumgebung Γ und einen in Γ typiserbaren Term e, was ist der *allgemeinste* Typ τ, so daß $\Gamma \vdash e : \tau$?

Wir zeigen nun, daß (3.) für F_1 positiv beantwortet werden kann. Wir machen das, indem wir einen Algorithmus angeben, der den allgemeinsten Typ eines Terms angibt, wenn der Term typisierbar ist, und der mit Fehler abbricht, wenn der Term nicht typisierbar ist.

Der zentrale Schritt des Algorithmus ist die Unifikation zweier Typen. Hier sind zunächst die benötigten Definitionen über Substitutionen und Unifikatoren. Wir nehmen überall an, daß Substitutionen Typvariablen auf Typen abbilden. Sei $\emptyset$ die leere Substitution, die jede Typvariable auf sich selbst abbildet.

Definition. Eine Substitution S ist *allgemeiner* als eine Substitution S', in Symbolen $S' \preceq S$, wenn es eine Substitution R gibt, so daß $S' = R \circ S$.

Definition. Eine Substitution U heißt *Unifikator* zweier Typen τ_1, τ_2, wenn $U\tau_1 = U\tau_2$.

Der allgemeinste Unifikator zweier Typen kann immer bestimmt werden, wenn er existiert:

Algorithmus 8.6 Berechnung des allgemeinsten Unifikators $mgu(\tau_1, \tau_2)$ zweier Terme τ_1, τ_2.

Eingabe: Zwei Terme τ_1, τ_2.

Ausgabe: Der allgemeinste Unifikator von τ_1 und τ_2, falls τ_1 und τ_2 unifizierbar sind. Fehlerabbruch sonst.

Verfahren:

$$
\begin{array}{lcl}
mgu & : & (Typ, Typ) \to Subst \\
mgu\,(\alpha, \alpha) & = & \epsilon \\
mgu\,(\alpha, \tau) \text{ where } \alpha \notin \mathrm{tv}\,\tau & = & [\tau/\alpha] \\
mgu\,(\tau, \alpha) \text{ where } \alpha \notin \mathrm{tv}\,\tau & = & [\tau/\alpha] \\
mgu\,(\tau_1 \to \tau_1', \tau_2 \to \tau_2') & = & \text{let } S_1 = mgu\,(\tau_1, \tau_2) \\
 & & \quad\;\; S_2 = mgu\,(S_1\tau_1', S_1\tau_2') \\
 & & \text{in } \; S_2 \circ S_1
\end{array}
$$

Der Typrekonstruktionsalgorithmus tp arbeitet inkrementell. Er erhält als Eingabe ein Typurteil $\Gamma \vdash e : \tau$, wo Γ eine Typothese und τ ein *Prototyp* ist. Sowohl Typothese wie auch Prototyp enthalten partielle Information. Die Typothese enthält Kontext-Information über die freien Variablen eines Ausdrucks. Entsprechend spiegelt der Proptotyp wider, was im Konext über den verlangten Typ des Ausdrucks bekannt ist. Der Algorithmus tp liefert zu einem gegebenen Typurteil $\Gamma \vdash e : \tau$ die allgemeinste Substitution S, so daß $S\Gamma \vdash e : S\tau$ im Typsystem von Abbildung 11.1 ableitbar ist. Wenn keine solche Substitution existiert, wird abgebrochen. Das ist immer dann der Fall, wenn im Unifikationsalgorithmus mgu keine der vier Alternativen anwendbar ist.

Algorithmus 8.7 Algorithmus zur Typrekonstruktion in F_1.

Eingabe: Ein Typurteil $\Gamma \vdash e : \tau$.

Ausgabe: Wenn es eine Substitution S gibt, so daß $S\Gamma \vdash e : S\tau$, die allgemeinste solche Substitution, ansonsten Fehlerabbruch.

Verfahren:

$$
\begin{array}{lcl}
tp & : & (\Gamma \vdash e : \tau) \to Subst \\
\\
tp\,(\Gamma \vdash x : \tau) & = & mgu\,(\Gamma(x), \tau) \\
\\
tp\,(\Gamma \vdash \lambda x.e : \tau) & = & \text{let } \alpha, \beta \quad \text{neue Typvariablen} \\
 & & \quad\; S_1 \;\; = mgu\,(\alpha \to \beta, \tau) \\
 & & \quad\; S_2 \;\; = tp\,(S_1\Gamma, x : S_1\alpha \vdash e : S_1\beta) \\
 & & \text{in } \; S_2 \circ S_1 \\
\\
tp\,(\Gamma \vdash e_1\,e_2 : \tau) & = & \text{let } \alpha \quad \text{eine neue Typvariable} \\
 & & \quad\; S_1 = tp\,(\Gamma \vdash e_1 : \alpha \to \tau) \\
 & & \quad\; S_2 = tp\,(S_1\Gamma \vdash e_2 : S_1\alpha) \\
 & & \text{in } \; S_2 \circ S_1
\end{array}
$$

Die nächsten zwei Sätze drücken die Korrektheit und Vollständigkeit des Algorithmus aus.

Satz 8.8 Wenn $tp\,(\Gamma \vdash e : \tau) = S$, so ist $S\Gamma \vdash e : S\tau$ ableitbar.

Satz 8.9 Sei $\mathcal{S} = \{S \mid S\Gamma \vdash e : S\tau\}$. Wenn $\mathcal{S} \neq \{\}$, so existiert eine Substitution S, so daß $tp\,(\Gamma \vdash e : \tau) = S$, und $S' \preceq S$, für alle $S' \in \mathcal{S}$.

8.3 Summe- und Produkttypen

Das Typsystem des letzten Abschnitts behandelte nur Funktionen. In diesem Abschnitt sind erweitern wir dieses Kalkül um weitere Typen. Ein Typ ist nun ein Element folgender Sprache

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{Typvariable} \\
 & \mid & \iota & \text{Grundtyp} \\
 & \mid & \tau \to \tau & \text{Funktionstyp} \\
 & \mid & \tau \times \tau & \text{Produkttyp} \\
 & \mid & \tau + \tau & \text{Summentyp}
\end{array}
$$

Dabei steht

$$(\times I)\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$(\times E)\quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ e : \tau_2}$$

$$(+I)\quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}\ e : \tau_1 + \tau_2}$$

$$(+E)\quad \frac{\Gamma \vdash d : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ d\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow e_1 \mid \mathsf{inr}\ x \Rightarrow e_2 : \tau}$$

Abbildung 8.2: Regeln für Produkt- und Summentypen

- ι für einen Grundtyp, der nicht weiter zerlegbar ist, wie z. B. int oder bool,

- $\tau_1 \times \tau_2$ für den Typ aller Paare (x, y), deren erste Komponenten vom Typ τ_1 sind und deren zweite Komponenten vom Typ τ_2 sind,

- $\tau_1 + \tau_2$ für die markierte (engl. "tagged") Vereinigung der Typen τ_1 und τ_2. Werte dieses Typs sind $\mathsf{inl}\, x$, wo $x : \tau_1$, und $\mathsf{inr}\, y$, wo $y : \tau_2$.

Produkttypen sind eine einfache Form von Records, während Summentypen als eine einfache Form von Varianten-Records (in Modula-2) oder Unions (in C) gesehen werden können. Anders als bei C-Unions ist allerdings durch die inl- und inr-Markierung immer klar, welche Variante definiert ist.

Basis-Operationen auf den neuen Typen sind wie folgt definiert.

- Für Grundtypen: Eine typabhängige Menge von Operationen, die alle in einer anfänglichen Typumgebung Γ_0 definiert werden. Eine typische Anfangsumgebung Γ_0 würde unter anderem die folgenden Bindungen enthalten:

$$\begin{array}{lcl}
0 & : & \mathsf{int}, \\
\mathsf{succ} & : & \mathsf{int} \to \mathsf{int}, \\
\mathsf{pred} & : & \mathsf{int} \to \mathsf{int}, \\
(+) & : & \mathsf{int} \times \mathsf{int} \to \mathsf{int}, \\
(-) & : & \mathsf{int} \times \mathsf{int} \to \mathsf{int}, \\
\mathsf{true} & : & \mathsf{bool}, \\
\mathsf{false} & : & \mathsf{bool}, \\
\mathsf{not} & : & \mathsf{bool} \to \mathsf{bool}, \\
(\mathsf{and}) & : & \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}, \\
(\mathsf{or}) & : & \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}.
\end{array}$$

- Für Produkttypen: Die beiden Selektoren fst und snd, die die erste bzw. zweite Komponente des Paares zurückgeben.

- Für Summentypen: Der case-Ausdruck

$$\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow e_1 \mid \mathsf{inr}\ x_2 \Rightarrow e_2.$$

Typenregeln für Konstruktoren und Selektoren sind in Abbildung 8.2 wiedergegeben.

Aufgabe 8.3 Erweitere Algorithmus 8.7 um Produkttypen und Summentypen.

8.4 Der Curry-Howard Isomorphismus

Beim Lesen des letzten Abschnitts haben Sie sich vielleicht gefragt, warum wir gerade Summen- und Produkttypen als Erweiterung von F_1 gewählt haben, und nicht etwas anderes, wie z. B. Mengentypen, Zeigertypen usw. Außerdem ist natürlich die Frage berechtigt, ob nicht statt den verwendeten Typregeln andere Typregeln vorzuziehen seien? Sind Typsysteme also nur beliebige Konstruktionen, die genausogut auch anders aufgebaut sein könnten? Sicher hat der Autor einer Programmiersprache die Freiheit, "sein" Typsystem den Bedürfnissen seiner Sprache anzupassen. Andererseits hat die beschriebene Formulierung von F_1 den Vorteil, daß sie enge Beziehungen zur mathematischen Logik hat.

Dies wird deutlich, wenn man alle Terme und Variablennamen in Abbildung 11.1 und Abbildung 8.2 wegläßt, und folgende Übersetzung vornimmt:

$$\begin{array}{ccc}
\to & \approx & \Rightarrow \\
\times & \approx & \wedge \\
+ & \approx & \vee.
\end{array}$$

$$
\begin{array}{ll}
\text{(TAUT)} & \Gamma, \tau \vdash \tau \\[1ex]
\text{(TRUE)} & \Gamma \vdash \mathit{true} \\[2ex]
(\wedge\text{-I}) & \dfrac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \wedge \tau_2} \\[2ex]
(\wedge\text{-E}) & \dfrac{\Gamma \vdash \tau_1 \wedge \tau_2}{\Gamma \vdash \tau_1} \qquad \dfrac{\Gamma \vdash \tau_1 \wedge \tau_2}{\Gamma \vdash \tau_2} \\[2ex]
(\vee\text{-I}) & \dfrac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 \vee \tau_2} \qquad \dfrac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \vee \tau_2} \\[2ex]
(\vee\text{-E}) & \dfrac{\Gamma \vdash \tau_1 \vee \tau_2 \qquad \Gamma, \tau_1 \vdash \tau \qquad \Gamma, \tau_2 \vdash \tau}{\Gamma \vdash \tau} \\[2ex]
(\Rightarrow\text{-I}) & \dfrac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \Rightarrow \tau_2} \\[2ex]
(\Rightarrow\text{-E}) & \dfrac{\Gamma \vdash \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2}
\end{array}
$$

Abbildung 8.3: Positive Aussagenlogik

Das resultierende System ist in Abbildung 8.3 gezeigt. Bemerkenswerterweise entsteht genau das Regelsystem zur Aussagenlogik von Abbildung 2.1!

Diese Übereinstimmung legt nahe, daß die Typen einer Programmiersprache den Aussagen einer Logik entsprechen — in diesem Falle den Aussagen der intuitionistischen Aussagenlogik. Wenn also Typ $\approx$ Aussage, welche Rolle spielt dann die Termkomponente eines Typurteils? Es stellt sich heraus, daß die Termkomponente als formalisierte Version des *Beweises* des Typs bzw. der Aussage gesehen werden kann. Im einzelnen:

- Um die Aussage $P \Rightarrow Q$ zu beweisen, benötigen wir ein Beweis-Schema, das, gegeben einen Beweis x von P, uns einen Beweis e von Q liefert. Dies wird durch das Funktions-Schema $\lambda x.e$ ausgedrückt.
- Nimm an, wir haben einen Beweis d der Aussage $P \Rightarrow Q$ und einen Beweis e der Aussage P. Dann erhalten wir einen Beweis von Q indem wir e in d einsetzen. Dies ist genau die Funktionsanwendung $d\,e$.
- Um einen Beweis der Aussage $P \wedge Q$ zu erhalten, benötigen wir einen Beweis d von P und einen Beweis e von Q, geschrieben (d, e).
- Wenn ein Beweis e von $P \wedge Q$ vorliegt, so gibt uns die erste Komponente $\mathsf{fst}\,e$ des Paars e einen Beweis für P, und die zweite Komponente gibt uns einen Beweis für Q.
- Um $P \vee Q$ zu beweisen, müssen wir entweder P oder Q beweisen[1] Wir kennzeichnen den Beweis noch, so daß klar ist welche Aussage bewiesen wurde, und schreiben $\mathsf{inl}\,e$ falls e P beweist, und $\mathsf{inr}\,e$, falls e Q beweist.
- Um einen Beweis e der Aussage $P \vee Q$ weiterzuverwenden, müssen wir eine Fallunterscheidung machen. Nimm an, wir haben zwei Beweisschemen d, e, die unter der Annahme, daß P bzw. Q bewiesen ist, eine andere Aussage R beweisen. Dann folgt R aus $P \vee Q$ mit dem Beweis
$$\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow d \mid \mathsf{inr}\ x \Rightarrow e.$$

Diese Analogie wird nach ihren Entdeckern Curry-Howard Isomorphismus genannt. Sie hat praktische Auswirkungen in automatischen Satzbeweisern. Betrachte zum Beipiel den folgenden Satz: Für beliebige Prädikate A, B, C,

$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \wedge B) \Rightarrow C). \tag{8.1}$$

[1] Das ist das kennzeichnende Charkteristikum der intuitiven Logik, im Gegensatz zur klassischen Logik, wo z.B. die Aussage $P \vee \neg P$ immer gilt, so daß weder P noch $\neg P$ bewiesen werden müßte ("tertium non datur").

Um einen Beweis dieses Satzes zu erhalten, ersetzen wir $\Rightarrow$ durch $\to$ und $\wedge$ durch $\times$:

$$(A \to B \to C) \to (A \times B) \to C. \tag{8.2}$$

Nun müßen wir nur noch eine Funktion dieses Typs finden, z. B.

$$\lambda f.\lambda x.f(\mathsf{fst}\, x, \mathsf{fst}\, y)$$

Damit haben wir gleichzeitig einen Beweis der Aussage (8.1) und ein Element des Typs (8.2) gefunden.

Eine andere Anwendung des Curry/Howard Isomorphismus liegt im Rechner-unterstützten Programmbeweis. Der Typ eines Programms übernimmt dabei die Rolle der Programmspezifikation. Das Programm ist dann identisch zum Beweis, daß die Spezifikation erfüllbar ist. Um nichttriviale Spezifikationen ausdrücken zu können, wird allerdings ein viel mächtigeres Typsystem benötigt als das, was wir bisher kennengelernt haben. Solche Typsysteme wurden von Martin Løf untersucht [18].

8.5 Das Hindley/Milner System

Trotz der Anwesenheit von Typvariablen ist F_1 *monomorph*, d. H. bei einem gegebenen Typurteil $\Gamma \vdash e : \tau$ haben e und jeder Unterausdruck von e genau einen Typ. Dies schränkt die Flexibilität der Programmiersprache erheblich ein.

Aufgabe 8.4 Betrachte die Kodierung $(\lambda f.(f\ \mathtt{true}, f\ 0))\,(\lambda x.x)$ des Programms $\mathsf{let}\ f = \lambda x.x\ \mathsf{in}\ (f\ \mathtt{true}, f\ 0)$. Überzeugen Sie sich, daß dieses Programm keinen Typ in F_1 hat.

Das Problem entsteht, da in einem Programm eine Typvariable nur höchstens einmal instanziert werden kann. In der folgenden Erweiterung von F_1 wird nun die mehrfache Instanzierung von Typen zugelassen, aber nur für Variablen, die in einem let-Ausdruck definiert werden.

Wir nehmen let-Ausdrucke zur Termsyntax dazu:

$$\text{Terme} \qquad e \ ::=\ x \mid \lambda x.e \mid e\,e' \mid \mathsf{let}\ x = d\ \mathsf{in}\ e$$

Die freien Typvariablen eines Typs können generisch gemacht werden, indem sie an einem $\forall$-Quantor gebunden werden. Den so entstandenen Ausruck nennen wir ein *Typschema* σ

$$\sigma \ ::=\ \forall\alpha.\sigma \mid \tau$$

Syntax

Typvariablen	α	
Typen	τ	$::= \ \alpha \mid \tau \to \tau'$
Typeschemen	σ	$::= \ \forall\alpha.\sigma \mid \tau$
Variablen	x	
Terme	e	$::= \ x \mid \lambda x.e \mid e\,e' \mid \mathsf{let}\ x = e\ \mathsf{in}\ e'$

Typregeln

(TAUT) $\quad \Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$

$$(\forall \mathrm{I}) \quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\alpha.\sigma} \ (\alpha \notin \mathrm{tv}(\Gamma))$$

$$(\forall \mathrm{E}) \quad \frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : [\tau/\alpha]\sigma}$$

$$(\to \mathrm{I}) \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$(\to \mathrm{E}) \quad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\,e' : \tau}$$

$$(\mathrm{LET}) \quad \frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau}$$

Abbildung 8.4: Hindley/Milner Typsystem

Wichtig dabei ist, daß Typschemen keine Typen sind, also alle $\forall$-Quantoren nur ganz außen in einem Typscheme vorkommen können.

Die Typregeln für das Hindley/Milner System finden sich in Abbildung 8.4. Zu den Regeln von F_1 kommen drei neue Regeln hinzu.

- Die Regel ($\forall$E) erlaubt es, eine gebundene Typvariable in einem Typschema zu einem beliebigen Typ zu instanzieren.

- Die Regel ($\forall$I) erlaubt es, eine Typvariable, die in der Typumgebung nicht vorkommt, generisch zu machen. Beispiel:

$$\frac{y : \beta \ \vdash \ \lambda x.x : \alpha \to \alpha}{y : \beta \ \vdash \ \lambda x.x : \forall \alpha.\alpha \to \alpha}$$

  Nicht erlaubt ist die Verallgemeinerung, wenn die Typvariable in der Umgebung vorkommt. Beispiel:

$$y : \beta \ \vdash \ \lambda x.y : \alpha \to \beta$$

  kann nicht abgeleitet werden zu

$$y : \beta \ \vdash \ \lambda x.y : \forall \beta.\alpha \to \beta.$$

  Das zweite Typurteil sagt nämlich, daß $\lambda x.y$ jeden beliebigen Resultatstyp haben kann — es ist äquivalent zum Typurteil

$$y : \beta \ \vdash \ \lambda x.y : \forall \gamma.\alpha \to \gamma.$$

  Der Zusammenhang zwischen dem Typ von y in der Typothese und dem Resultatstyp von $\lambda x.y$ ginge also verloren.

- Die Regel für let-Ausdrücke besagt, daß der Typ von let $x = d$ in e gleich dem Typ von e ist, wenn x in der Typothese das das Typschema von d erhält. Man bemerke, daß alle anderen Regeln nach wie vor nur mit Typen, nicht mit Typschemen arbeiten.

Die Regeln für Produkt- und Summentypen sind dieselben wie für F_1. Sie werden hier nicht noch einmal aufgeführt.

Die operationelle Semantik von HM wird durch folgende Reduktionsregeln erklärt:

$$\begin{array}{lllll}
\beta: & (\lambda x.d)\,e & \to & [e/x]d \\
let: & \mathsf{let}\ x = d\ \mathsf{in}\ e & \to & [d/x]e \\
\delta: & p\,v & \to & \delta(p,v) \\
\pi_1: & \mathsf{fst}(d,e) & \to & d \\
\pi_2: & \mathsf{snd}(d,e) & \to & e \\
case_1: & \mathsf{case}\ (\mathsf{inl}\ d)\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow e \mid \ldots & \to & [d/x]e \\
case_2: & \mathsf{case}\ (\mathsf{inr}\ d)\ \mathsf{of}\ \ldots \mid \mathsf{inr}\ x \Rightarrow e & \to & [d/x]e
\end{array}$$

Die δ Regel ist hier ein Schema, das die Auswertung von primitiven Operationen auf Grundtypen angibt. Über die Funtkion $\delta(\cdot,\cdot)$ ist nichts ausgesagt, außer, daß wir annehmen, daß δ überall definiert und typkorrekt ist. D.h., für jede Typumgebung Γ, Operator p, Wert v, wenn $\Gamma \vdash p : \tau_1 \to \tau_2$ und $\Gamma \vdash v : \tau_1$, dann $\Gamma \vdash \delta(p,v) : \tau_2$.

Satz 8.10 (Typ-Erhaltung von Reduktion in HM) Wenn $\Gamma \vdash e : \tau$ und $e \to d$ dann gilt auch $\Gamma \vdash d : \tau$.

Satz 8.10 gibt eine wichtige semantische Eigenschaft des Typsystems von F_1 an. Wenn ein Term e einen Typ τ hat, so wissen wir, daß das Endresultat der Auswertung von e wieder den Typ τ hat.

Das folgende Lemma gibt uns weiteren Aufschluß über die Zusicherungen, die in einer Typisierung enthalten sind.

Lemma 8.11 Sei e eine Normalform, die kein Wert ist. Dann ist e in HM nicht typisierbar.

Aus Satz 8.10 und Lemma 8.11 folgt nun, daß das Hindley/Milner System Typfehler zur Laufzeit vollständig ausschließt:

Satz 8.12 Wenn $\Gamma \vdash e : \tau$ und die Auswertung von e terminiert, so endet diese Auswertung in einem Wert vom Typ τ.

Beweis: Nimm an, $\Gamma \vdash e : \tau$, $e \twoheadrightarrow e'$ und e' ist eine Normalform. Mit Satz 8.10 gilt $\Gamma \vdash e' : \tau$. Mit Lemma 8.11 gilt, daß e' ein Wert ist. □

Satz 8.12 begründete den Slogan daß wohlgetypte Programme nicht "schief gehen" können. Dabi muß allerdings auf das "Kleingedruckte" geachtet werden. Damit der Satz 8.12 gilt, müssen primitive Operationen immer überall in ihrem Geltungsbereich definiert sein — dies folgt aus den Bedingungen, die wir an die δ-Funktion gestellt haben. Zum Beispiel müßte der Ausdruck $x/0$ einen Wert zurückgeben. Eine Möglichkeit, die z. B. in ML verwirklicht ist, ist es, Ausnahmen als Werte zuzulassen. Dann ist $\delta((/),(x,0)) = \mathsf{ArithmeticError}$, was als zulässiger Wert interpretiert wird.

Das Hindley-Milner System hängt eng mit dem einfach getypten λ-Kalkül zusammen. Es gilt nämlich:

Satz 8.13 Sei Γ eine *monomorphe* Typumgebung von Bindungen $x_1 : \tau_1, \ldots, x_n : \tau_n$. Sei * die Abbildung, die jeden let-Ausdruck $\mathsf{let}\ x = d\ \mathsf{in}\ d'$ durch $[d/x]d'$ ersetzt. Dann gilt

$$\Gamma \vdash_{HM} e : \tau \qquad \Leftrightarrow \qquad \Gamma \vdash_{F_1} e^* : \tau.$$

Es folgt, da das Hindley/Milner System genau wie das einfach getypte λ-Kalkül die starke Normalisierungseigenschaft besitzt. Deshalb ist auch Rekursion nicht ausdrückbar. Dieses Problem kann aber behoben werden, indem ein Fixpunktoperator (mit Typschema $\forall\alpha.(\alpha \to \alpha) \to \alpha$) in der initialen Typumgebung postuliert wird.

8.6 Der typisierte Lambda-Kalkül zweiter Ordung

fehlt noch.

8.7 Überladung und Records

Siehe dazu das Papier *A Second Look at Overloading*.

Kapitel 9

Modelle von funktionalen Programmen

Wir definieren Modelle zunächst für deterministische Sprachen, für die eine partielle Auswertungsfunktion $eval$ von Termen nach Werten definiert ist. Z. B. ist das reine Lambda-Kalkül eine solche Sprache, wo $eval(e)$ derjenige Wert v ist, so daß $e \mapsto\!\!\!\to v$, wenn ein solcher Wert existiert.

Definition. Sei L eine Sprache, deren operationelle Bedeutung durch eine Reduktionsrelation $\to$ gegeben ist. Ein *Modell* von L ist ein Paar $(D, [\![\cdot]\!])$, wo

- D eine beliebige Menge ist,
- $[\![\cdot]\!]$ eine Funktion von Termen der Sprache nach D ist, so daß immer wenn $d \to e$ gilt, auch gilt, daß $[\![d]\!] = [\![e]\!]$.

Eine Interpretationsfunktion ist *kompositionell*, wenn für alle Terme e der Form $F(e_1, \ldots, e_n)$, wo $n \geq 0$ und F ein beliebiges syntaktisches Konstrukt ist, gilt: Es existiert eine Funktion f, so daß

$$[\![e]\!] \quad = \quad f([\![e_1]\!], \ldots, [\![e_n]\!]).$$

Kompositionelle Interpretationen lassen strukturelle Induktion über Bedeutungen als Beweisprinzip zu, und sind deshalb sehr gut zu handhaben. Wir werden im folgenden nur kompositionelle Interpretationen behandeln.

Beispiel 9.1

1. $(\{a\}, [\![\cdot]\!])$ mit $[\![e]\!] = a$ ist ein Modell jeder deterministischen Sprache. Dieses Modell heißt das *triviale Modell*.

2. Sei $(=)$ die kleinste Äquivalenzrelation, die $(\longrightarrow)$ enthält. Sei $\overline{e} = \{e' \mid e = e'\}$ die Äquivalenzklasse von e bzgl. $(=)$. Dann ist $(\{e \mid e \in L\}, [\![\cdot]\!])$ mit $[\![e]\!] = \overline{e}$ ein Modell von L, das das *Termmodell* von e genannt wird.

Modelle legen Gleichheitsrelationen auf Termen fest. Gegeben ein Modell $(D, [\![\cdot]\!])$, sei

$$d =_{(D,[\![\cdot]\!])} e \qquad \text{falls} \qquad [\![d]\!] = [\![e]\!].$$

Verschiedene Modelle für eine Sprache können über die Gleichheitsrelationen, die sie induzieren, verglichen werden. Wir sagen, ein Modell I ist *feiner* als ein Modell J, wenn für alle Terme d, e gilt: Wenn $d =_I e$ dann auch $d =_J e$. In Symbolen: $=_I \subseteq =_J$.

Offensichtlich ist das Termmodell das feinste Modell einer jeden Sprache, und das triviale Modell ist das gröbste. Beide Modelle sind aber in gewisser Hinsicht uninteressant, da sie uns keinen Aufschluß über die beabsichtigte Bedeutung von Termen geben. Ein weiteres Modell, das in seiner Mächtigkeit zwischen dem Termmodell und dem trivialen Modell liegt und das uns genaueren Aufschluß über die beabsichtigte Bedeutung von Termen gibt, wird im nächsten Abschnitt behandelt.

9.1 Das Fixpunktmodell von PCF

In diesem Abschnitt behandeln wir ein Modell für eine einfache getypte funktionale Sprache, deren Syntax und Semantik in Abbildung 9.1 bis Abbildung 9.3 gegeben sind. Die Sprache besteht aus Funktionen mit Namensauswertung und Konstanten und primitiven Operatoren für natürliche Zahlen und Wahrheitswerte sowie einem Fixpunktoperator. In dem verwendeten Typsystem ist Church-Kodierung nicht möglich, deshalb werden Konstanten und primitive Operatoren explizit in die Sprache aufgenommen.

Die Reduktionsrelation für PCF hat die folgenden zwei nützlichen Eigenschaften:

Satz 9.2 $\longrightarrow$ ist konfluent.

Satz 9.3 Die Werte von PCF sind abgeschlossen bzgl. $\longrightarrow$: Wenn $v \longrightarrow e$, so ist e wieder ein Wert.

$$\begin{array}{llll}
\text{Typen} & \tau & ::= & \mathsf{Nat} \mid \mathsf{bool} \mid \tau \longrightarrow \tau \\
\text{Variablen} & x & \in & \mathcal{V} \\
\text{Terme} & e & ::= & 0 \mid \mathsf{succ}\, e \mid \mathsf{pred}\, e \mid \mathsf{iszero}\, e \\
 & & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \\
 & & \mid & x \mid \lambda x : \tau . e \mid e_1\, e_2 \mid \mathsf{fix}\, e \\
\text{Werte} & v & ::= & 0 \mid \mathsf{succ}\, v \\
 & & \mid & \mathsf{true} \mid \mathsf{false} \\
 & & \mid & x \mid \lambda x : \tau . e
\end{array}$$

$$\Gamma \vdash 0 : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool}$$

$$\frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\, e : \mathsf{Nat}} \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}\, e : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau}$$

$$\frac{\Gamma . x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau . e) : \tau \longrightarrow \tau'} \qquad \frac{\Gamma \vdash f : \tau \longrightarrow \tau' \quad \Gamma \vdash e : \tau}{\Gamma \vdash f\, e : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \longrightarrow \tau}{\Gamma \vdash \mathsf{fix}\, e : \tau}$$

Abbildung 9.1: Syntax von PCF

Reduktionsregeln

$$\begin{array}{rcl}
\mathsf{pred}(\mathsf{succ}\, v) & \to & v \\
\mathsf{iszero}\, 0 & \to & \mathsf{true} \\
\mathsf{iszero}\, (\mathsf{succ}\, v) & \to & \mathsf{false} \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \to & e_1 \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \to & e_2 \\
(\lambda x.d)\, e & \to & [e/x]d \\
\mathsf{fix}\, (\lambda x.e) & \to & [\mathsf{fix}\, (\lambda x.e)/x]e.
\end{array}$$

Reduktionsrelation = Reduktionsregegeln +

$$\frac{d \to e}{C[d] \to C[e]}$$

Abbildung 9.2: Operationale Semantik von PCF

Die Bedeutung von Typen und Termen ist in Abbildung 9.3 erklärt. Der Zielbereich der Interpretation $[\![\cdot]\!]$ ist hier eine Funktion von einer *Umgebung* η in den Wertebereich D. D ist die Vereinigung folgender Bereiche:

- Der Bereich $nat_\perp$, der aus den natürlichen Zahlen und dem Wert $\perp_{nat}$ besteht.
- Der Bereich $bool_\perp$, der aus den Wahrheitswerten T, F und dem Wert $\perp_{bool}$ besteht.
- Für jedes Paar von Bereichen X, Y den Bereich $X \to Y$, der aus den stetigen Funktionen von dem Bereich X zu dem Bereich Y bestehen. Kleinstes Element $\perp_{X \to Y}$ des Bereichs $X \to Y$ ist die Funktion, die jedem $x \in X$ den Wert $\perp_Y$ zuordnet.

Wir lassen den Index von $\perp$ oft weg, wenn er aus dem Kontext ersichtlich ist.

Eine Umgebung η ist eine Funktion von Variablen nach Elementen von D. $\eta[x := v]$ bezeichnet die Erweiterung von η um die Bindung von x an v.

$$\eta[x := v]\, y \;=\; \begin{cases} v & \text{falls } x = y \\ \eta\, y & \text{sonst} \end{cases}$$

Der nächste Satz sagt aus, daß die Interpretationsfunktion typkonform ist.

Bereiche

$$\begin{array}{rcl}
[\![\mathsf{Nat}]\!] & = & Nat_\perp \\
[\![\mathsf{bool}]\!] & = & Bool_\perp \\
[\![\tau_1 \to \tau_2]\!] & = & [\![\tau_1]\!] \to [\![\tau_2]\!]
\end{array}$$

Bedeutung von Termen

$$\begin{array}{lcl}
[\![x]\!]\eta & = & \eta(x) \\
[\![0]\!]\eta & = & 0 \\
[\![\mathsf{true}]\!]\eta & = & T \\
[\![\mathsf{false}]\!]\eta & = & F \\
[\![\mathsf{succ}\, e]\!]\eta & = & \begin{cases} \perp & \mathsf{if}\ [\![e]\!]\eta = \perp \\ x+1 & \mathsf{if}\ [\![e]\!]\eta = x \in Nat \end{cases} \\
[\![\mathsf{pred}\, e]\!]\eta & = & \begin{cases} \perp & \mathsf{if}\ [\![e]\!]\eta \in \{0, \perp\} \\ x-1 & \mathsf{if}\ [\![e]\!]\eta = x > 0 \end{cases} \\
[\![\mathsf{iszero}\, e]\!]\eta & = & \begin{cases} \perp & \mathsf{if}\ [\![e]\!]\eta = \perp \\ T & \mathsf{if}\ [\![e]\!]\eta = 0 \\ F & \mathsf{if}\ [\![e]\!]\eta > 0 \end{cases} \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\eta & = & \begin{cases} \perp & \mathsf{if}\ [\![e_1]\!]\eta = \perp \\ [\![e_2]\!]\eta & \mathsf{if}\ [\![e_1]\!]\eta = \mathsf{true} \\ [\![e_3]\!]\eta & \mathsf{if}\ [\![e_1]\!]\eta = \mathsf{false} \end{cases} \\
[\![\lambda x.e]\!]\eta & = & \lambda y.[\![e]\!]\eta[x := y] \\
[\![f\ e]\!]\eta & = & ([\![f]\!]\eta)\, [\![e]\!]\eta \\
[\![\mathsf{fix}\, e]\!]\eta & = & fix\, ([\![e]\!]\eta)
\end{array}$$

Abbildung 9.3: Ein Modell für PCF

Satz 9.4 Sei $\Gamma \vdash e : \tau$ ein Typurteil, und sei η eine Umgebung, so daß für alle $x : \tau' \in \Gamma$ gilt, daß $[\![x]\!]\eta \in [\![\tau']\!]$. Dann ist $[\![e]\!]\eta \in [\![\tau]\!]$.

Der folgende Satz impliziert, daß $(D, [\![\cdot]\!])$ ein Modell der Sprache PCF ist.

Satz 9.5 Wenn $d \to e$ für zwei PCF-Terme d, e, so ist $[\![d]\!] = [\![e]\!]$.

9.2 Operationelle Äquivalenz

In den beiden letzten Abschnitten haben wir drei Modelle für PCF kennengelernt. Es stellt sich nun die Frage, ob eines dieser drei Modelle in irgendeiner Hinsicht besser ist als die beiden anderen, und allgemeiner, ob es ein "bestes" Modell gibt. Sicher entspricht das Fixpunktmodell eher unserem intuitiven Verständnis der Sprache PCF, aber wir hätten gerne eine präzisere Charakterisierung der Güte eines Modells. Einen Hinweis bietet uns die Überlegung, daß ein Modell vorhersagen sollte, ob zwei Terme in jedem beliebigen Programm-Kontext austauschbar sind. Terme, die austauschbar sind, werden auch *operationell äquivalent* genannt.

Definition. Zwei Terme d, e sind *operationell äquivalent*, in Symbolen $d \cong e$, wenn für jeden Kontext C, so daß $C[d]$ und $C[e]$ abgeschlossen sind, gilt:

$$C[d]\Downarrow \quad \text{genau dann, wenn} \quad C[e]\Downarrow\,.$$

Intuitiv gesehen sind Kontexte C mögliche Experimentieranordnungen, in die die Terme d, e eingebettet werden können. Die einzige mögliche Beobachtung eines Experiments ist, ob ein Term sich zu einer schwachen Normalform reduziert. Zwei Terme sind operationell äquivalent, wenn sie in gleichen Experimentieranordnungen immer gleiche Resultate liefern.

Offensichtlich ist operationelle Äquivalenz diejenige Gleichheitsrelation auf Termen, die den Programmentwickler hauptsächlich interessiert, da sie ihm sagt, ob zwei Programmstücke (z. B. verschiedene Implementierungen einer Bibliotheksfunktion) in jedem beliebigen Programm austauschbar sind. Allerdings ist operationelle Äquivalenz oft sehr schwierig nachzuweisen, da sie eine Argumentation über alle möglichen Programmkontexte erfordert.

Ein Modell ist um so "besser", je genauere Voraussagen es uns über operationelle Äquivalenz von Termen bietet. Dabei sollen alle Voraussagen konservativ sein. D. H. wenn zwei Terme die gleiche Bedeutung in einem Modell haben, dann sollen sie auch operationell äquivalent sein. Die Umkehrung muß nicht unbedingt gelten. Wir definieren also:

Definition. Ein Modell $(D, [\![\cdot]\!])$ ist *adäquat* für eine Sprache mit operationeller Äquivalenzrelation $\cong$, wenn $=_{[\![\cdot]\!]} \subseteq \cong$.

Definition. Ein Modell $(D, [\![\cdot]\!])$ ist *abstrakt* für eine Sprache mit operationeller Äquivalenzrelation $\cong$, wenn $=_{[\![\cdot]\!]} = \cong$.

Satz 9.6 Das Termmodell von PCF ist adäquat.

Beweis: Es sei im Termmodell $[\![d]\!] = [\![e]\!]$. Wir zeigen, daß $d \cong e$. Sei C ein Kontext, so daß $C[d]$ und $C[e]$ abgeschlossen sind und $C[d]\Downarrow$. Dann gibt es einen Wert v, so daß $C[d] \twoheadrightarrow v$. Da $[\![\cdot]\!]$ kompositionell ist, ist $[\![C[d]]\!] = [\![C[e]]\!]$. Wegen der Definition von $[\![\cdot]\!]$ ist deshalb

$$\overline{C[d]} = \overline{C[e]} = \overline{v}$$

also $C[e] = v$. Da $(\to)$ konfluent ist, gibt es einen Term f, so daß $C[e] \twoheadrightarrow f$ und $v \twoheadrightarrow f$. Da Werte abgeschlossen sind bzgl. Reduktion, ist f ein Wert. Es folgt, daß $C[e]\Downarrow$. Da C beliebig war, gilt $d \cong e$. □

Satz 9.7 Das triviale Modell von PCF ist nicht adäquat.

Beweis: $\mathsf{true} \not\cong \mathsf{false}$, aber $[\![\mathsf{true}]\!] = [\![\mathsf{false}]\!]$ im trivialen Modell. □

Aufgabe 9.1 Ist das Termmodell von PCF abstrakt? Wie steht es mit dem trivialen Modell?

9.3 Abstraktheit und PCF

Wir betrachten nun das Fixpunktmodell von PCF unter den im letzten Abschnitt aufgestellten Kriterien.

Satz 9.8 Das Fixpunktmodell von PCF ist adäquat.

Beweis: Siehe [14] oder [5] □

Ist das Fixpunktmodell von PCF abstrakt? Interessanterweise heißt die Antwort nein. Der Grund dafür liegt darin, daß es Funktionen in dem Modell gibt, die nicht Termen entsprechen, die in der Sprache PCF definierbar sind. Eine solche Funktion ist die *parallele Disjunktion*, *por*, die wie folgt definiert ist:

$$por\ x\ y = \begin{cases} T & \text{wenn } x = T \vee y = T \\ F & \text{wenn } x = F \wedge y = F \\ \perp & \text{sonst} \end{cases}$$

Hier sind die Wahrheitstabellen der paralellen Disjunktion *por* und der sequentiellen Disjunktion *or*, die in PCF durch

$$or\ x\ y = \mathsf{if}\ x\ \mathsf{then\ true\ else}\ y$$

definierbar ist.

x y	$\perp$	F	T
$\perp$	$\perp$	$\perp$	T
F	$\perp$	F	T
T	T	T	T

por x y

x y	$\perp$	F	T
$\perp$	$\perp$	$\perp$	$\perp$
F	$\perp$	F	T
T	T	T	T

or x y

Die beiden Operatoren unterscheiden sich insofern, als

$$por \perp T = T \quad \text{und} \quad or \perp T = \perp.$$

Wir definieren nun zwei Terme

$$X, Y : (\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool} \tag{9.1}$$

so daß $X(por) \not\cong Y(por)$, aber für alle Funktionen f, die in PCF definiert werden können:

$$X(f) \cong Y(f). \tag{9.2}$$

Dann gilt wegen (9.1) und der Adäquatheit von $[\![\cdot]\!]$, daß $[\![X]\!] \neq [\![Y]\!]$. Umgekehrt gilt wegen (9.2) und dem folgenden *Kontextlemma* (Lemma 9.9), daß $X \cong Y$.

Lemma 9.9 Für alle PCF-Terme $e_1, e_2 : \tau_1 \to \tau_2$,

$$e_1 \cong e_2 \quad \Leftrightarrow \quad \forall d : \tau_1 . e_1\, d \cong e_2\, d.$$

Es bleibt noch, Terme X, Y zu finden, für die (9.1) und (9.2) gelten. Sei dazu `Omega` ein Boolescher Term, so daß in jeder Umgebung η gilt: $[\![\mathtt{Omega}]\!]\eta = \perp$. Eine mögliche Definition wäre z. B.

```
Omega = fix (lambda x: Bool → Bool. x)
```


Dann definieren wir:

```
X = lambda f: Bool → Bool → Bool.
        if f true Omega then
          if f Omega true then
            if f false false then Omega else true
          else Omega
        else Omega

Y = lambda f: Bool → Bool → Bool.
        if f true Omega then
          if f Omega true then
            if f false false then Omega else false
          else Omega
        else Omega
```

Wir berechnen nun die Bedeutung von X und Y im Fixpunktmodell. Sei die Bedingung $por?(f)$ wie folgt definiert:

$$por?(f) \quad = \quad f\, T \perp = T \ \wedge\ f \perp T = T \ \wedge\ f\, F\, F = F$$

Sei η eine beliebige Umgebung. Einsetzen der Bedeutung von X und Y und Vereinfachen ergibt:

$$[\![X]\!]\eta \quad = \quad \lambda f. \begin{cases} T & \text{wenn } por?(f) \\ \perp & \text{sonst} \end{cases}$$

$$[\![Y]\!]\eta \quad = \quad \lambda f. \begin{cases} F & \text{wenn } por?(f) \\ \perp & \text{sonst.} \end{cases}$$

Einsetzen und Ausrechnen ergibt weiter, daß $por?(por)$ gilt. Also gilt

$$([\![X]\!]\eta)\, por \quad = \quad T \quad \neq \quad F \quad = \quad ([\![Y]\!]\eta)\, por$$

und deshalb $[\![X]\!] \neq [\![Y]\!]$. Umgekehrt kann für jede PCF-definierbare Funktion f gezeigt werden, daß $por?(f)$ nicht gültig ist. Intuitiv folgt das, da Kaskaden von bedingten Anweisungen die einzige Möglichkeit in PCF sind, um Boolesche Funktionen programmieren. Jede solche Kaskade legt aber eine Reihenfolge fest, in der die Operanden der Funktion untersucht werden. Im Gegensatz dazu ist die einzige mögliche Implementierungsstrategie für *por* die *parallele* Auswertung beider Operanden, verbunden mit einem Abbruch der Auswertung eines Operanden, falls der andere Operand `true` ergibt. Also gilt für jede PCF-definierbare Funktion f:

$$[\![X\ f]\!]\eta = \perp = [\![Y\ f]\!]\eta.$$

Wegen der Adäquatheit des Modells folgt damit:

$$X\ f \cong Y\ f$$

und damit (9.2). Damit ist bewiesen, daß $[\![\cdot]\!]$ nicht abstrakt ist.

Die Diskrepanz zwischen Gleichheit im Modell und operationeller Äquivalenz läßt sich auf zwei Arten beheben. Einerseits kann man die "Schuld" auf die Sprache statt das Modell schieben und argumentieren, daß in PCF der *por*-Operator fehlt. Plotkin hat gezeigt, daß damit tatsächlich Abstraktheit erreicht wird [14]. Sei PCF+por die Sprache, die entsteht, wenn man den por Operator in der oben beschriebenen Semantik zu PCF dazunimmt.

Satz 9.10 Das Fixpunktmodell ist abstrakt für PCF+por.

Allerdings ist die Erweiterung um *por* wenig praktikabel, da *por* eine aufwendige parallele Suche erfordert, die im Gegensatz zur ansonsten sequentiellen Auswertung von PCF steht.

Ein anderer Lösungsansatz besteht darin, die Sprache PCF unverändert zu lassen, aber ein abstrakteres Modell als das Fixpunktmodell zu finden. Milner hat gezeigt, daß das immer möglich ist. Man nehme einfach als Bedeutungsbereich die operationellen Äquivalenzklassen $\tilde{e}$ von Termen e,

$$\tilde{e} \quad = \quad \{d \mid e \cong d\}$$

und definiere

$$[\![e]\!] = \tilde{e}$$

Aufgabe 9.2 Zeige, daß das Milnersche Modell abstrakt ist.

Auch wenn das Milnersche Modell abstrakt ist, ist es dennoch nicht sehr nützlich. Zum einen gibt es uns keine Intuition, was Terme in PCF bedeuten. Zum anderen gibt es uns keine Hilfestellung zum Nachweis von operationellen Äquivalenzen — um eine Gleichung $[\![d]\!] = [\![e]\!]$ nachzuweisen, muß ja gerade $d \cong e$ gezeigt werden.

Die Frage, ob es abstrakte Modelle für sequentielles PCF gibt, die nichttrivial sind, d. H. die sich nicht über operationelle Äquivalenz definieren, hat die Semantiker seit 1977 nicht mehr losgesgelassen. 1994 wurden dann sogar zwei solche Modelle gefunden, und zwar von den Gruppen von Abramsky [1] und Ong. Beide Modelle bilden PCF-Terme in Spiele einer gegebenen Spieltheorie mit gewissen Eifpgenschaften ab. Beide Modelle sind relativ komplex. Für den "täglichen Gebrauch" ist es deshalb sicher am einfachsten, das Fixpunktmodell weiterzuverwenden, auch wenn es nicht abstrakt ist.

Kapitel 10

Funktionales Programmieren im Großen

In den letzten Kapiteln haben wir gesehen, daß sich mit funktionalen Programmen Algorithmen und Datenstrukturen in klarer und oft überraschend knapper Weise ausdrücken lassen. Jeder, der versucht hat, realistische Programme funktional zu schreiben, wird aber bestätigen, daß dies für größere Systeme erheblich schwieriger wird.

Beispiel 10.1 Um den Typrekonstruktionsalgorithmus (Algorithmus 8.6) rein funktional zu implementieren, müßte man zwei Probleme lösen. Wie lassen sich neue Typvariablen schaffen? Und wie kann man im Fehlerfall abbrechen? Ohne Seiteneffekte und Ausnahmen ist es das natürlichste, einen Zähler, der die Anzahl der "verbrauchten" Variablennamen zählt, jeder Funktion als Argument und als Resultat mitzugeben. Ebenfalls muß das Resultat einen Hinweis enthalten, ob ein Fehler entdeckt wurde.

Außerdem hat Algorithmus 8.6 ein Effizienzproblem, weil in fast jedem Schritt eine Substitution auf eine Typumgebung angewandt wird, was vermutlich mindestens Rechenzeit proportional zur Größe der Typumgebung brauchen wird. Eine Effizienzsteigerung kann erzielt werden, wenn die Anwendung einer Substitution verzögert wird, bis eine Typvariable tatsächlich dereferenziert werden muß. Also sollte auch die gegenwärtig gültige Substitution als Ein- und Ausgabe aller Funktionen auftauchen.

Eine Implementierung des Typprüfungsalgorithmus könnte dann wie folgt aussehen.

Typen:

```
type TVar   = Int
type Var    = Int
type Type   = TV TVar | Type ≻ Type
type Term   = Id Var | Lam Var Term | App Term Term
type Typo   = [(TVar, Type)]
type E a    = Suc a | Err String
type Subst  = TVar → Type
```

Operationen auf Substitutionen:

```
deref : Type → Subst → Type
deref (TV a) s =
  case s a of
    TV b where a /= b ⇒ deref (TV b) s
deref t s = t

assign : TVar → Type → Subst → Subst
assign a t s = λb. if a == b then t else s b
```

Operationen auf Typothesen:

```
insert : Typo → (Var, Type) → Typo
insert T (x, t) = (x, t) : T

lookup : Typo → Var → E Type
lookup [] y =
  Err "undeclared variable"
lookup ((x,t):xs) y =
  if x == y then Suc t
  else lookup xs y
```

Unifikation:

```
occurs : TVar → Type → Subst → Bool
occurs a t s =
  case deref t s of
    TV b ⇒
```



```
      a == b
    t1 ≻ t2 ⇒
      occurs a t1 s || occurs a t2 s

unify: (Type, Type) → Subst → E Subst
unify (t1, t2) s =
  case (deref t1 s, deref t2 s) of
    (TV a, TV b) where a == b ⇒
      Suc s
    (TV a, t) where not occurs a t ⇒
      Suc (assign a t s)
    (t, TV a) where not occurs a t ⇒
      Suc (assign a t s)
    (s1 ≻ t1, s2 ≻ t2) ⇒
      case unify s1 s2 s of
        Suc s'   ⇒ unify t1 t2 s'
        Err msg  ⇒ Err msg
    _ ⇒
      Err "types incompatible"
```

Typrekonstruktion:

```
tp : (Typo, Term, Type) → (Int, Subst) → E (Int, Subst)

tp (T, Id x, t) (n, s) =
  case lookup T x of
    Suc t' ⇒
      case unify (t, t') s of
        Suc s'   ⇒ Suc (n, s')
        Err msg  ⇒ Err msg
    Err msg' ⇒ Err msg'

tp (T, Lam x e, t) (n, s) =
  let a = TV (n)
      b = TV (n+1)
  in  case unify (a ≻ b, t) of
        Suc s' ⇒
          tp (insert T (x, a), e, b) (n+2, s')
        Err msg ⇒
          Err msg

tp (T, App d e, t) (n, s) =
```

```
let a = TV (n)
in  case tp (T, d, a ≻ t) (n+1, s) of
      Suc (n', s') ⇒
         tp (T, e, a) (n', s')
      Err msg ⇒
         Err msg
```

Diese Implementierung ist aus mehreren Gründen etwas unschön:

- Alle Funktionen haben lange Listen von Argumenten und Resultaten. Viele dieser Argumente und Resultate übernehmen die Rolle von globalen Variablen in einer Sprache mit Seiteneffekten. Auch wenn diese nur an wenigen Stellen im Programm gelesen oder verändert werden, so müssen sie doch durch alle Funktionsaufrufe "hindurchgereicht" werden. Dies ist nicht nur unschön, sondern auch gefährlich: Es ist leicht möglich, einer Funktion aus Versehen das falsche Argument zu übergeben.
- Es sind viele Fallunterscheidungen nötig, um im Fehlerfall abbrechen zu können. Auch wenn Fehler nur an wenigen Stellen im Programm auftreten können, muß doch fast jeder Funktionsaufruf an eine solche Fallunterscheidung gekoppelt werden. In einer Sprache mit Ausnahmen könnte dies vermieden werden.
- Die explizite Behandlung von Zustand und Ausnahmen an jedem Funktionsaufruf erschwert die Änderung des Programms.

Aufgabe 10.1 Welche Änderungen sind nötig, wenn im Fehlerfall nicht abgebrochen werden soll, sondern alle Fehlermeldungen in einer Liste zusammengefaßt werden?

Das Problem liegt darin, daß eine Einführung einer neuen "globalen Variable" die Änderung *aller* Funktionen erforderlich macht, da die Variable durch alle Funktionsaufrufe hindurchgereicht werden muß. Dies kann zwar dadurch notationell erträglich gehalten werden, daß alle solchen Argumente in einer Record-Struktur gehalten werden, aber dann tritt dafür ein Effizienzproblem auf, weil die Änderung eines Records in einer rein funktionalen Sprache eine Kopie des Records anlegt, was bei großen Records teuer wird.

Wir sehen also, daß die Eliminierung der "referentiell undurchsichtigen" globalen Variablen, Seiteneffekte und Ausnahmen ein zweischneidiges Schwert ist. Im Prinzip hilft das Explizieren aller Vorgänge der Verständlichkeit eines Programms. Andererseits aber wird das Programm dadurch manchmal erheblich verkompliziert, was wiederum seine Verständlichkeit verschlechtert.

Eine Lösung des Problems kann gefunden werden, wenn wir imperative Programme als funktionale Programme betrachten. Daß dies im Prinzip möglich ist, hat ja schon die denotationelle Semantik von IMP gezeigt, die ein imperatives Programm in eine funktionale Sprache übersetzt hat. In dieser Semantik werden alle Variablen zu einem *Zustand* S zusammengefaßt. Eine Anweisung ist eine Funktion von S nach S. Das Semikolon drückt die (funktionale) Komposition zweier Anweisungen aus.

Die Idee ist nun, statt "Zustand", "Anweisung", "sequentielle Komposition" vorzudefineren, diese vom Benutzer entsprechend seinen Anforderungen definieren zu lassen. Der Benutzer könnte z. B. die sequentielle Komposition so definieren, daß im Fehlerfall abgebrochen wird, und hätte so die Grundlagen einer Ausnahmebehandlung geschaffen. Ein Problem entsteht noch beim Lesen von Zustandsdaten. In der denotationalen Semantik von IMP wird das so gelöst, daß der Zustand als Argument an alle Teile eines Ausdrucks mitgegeben wird. Dies ist eine nicht-lokale Änderung des Programms und sollte deshalb vermieden werden. Statt dessen lassen wir alle Anweisungen ein Resultat zurückgeben. Dann ist es möglich, eine Lese-Anweisung zu definieren, deren Resultatsteil das gelesene Datum ist.

Der Spielraum des Programmierers bei der Definition von "Anweisung", "Zustand" usw. ist natürlich recht groß. Dennoch lassen sich alle solchen Definitionen in eine Struktur fassen, die im nächsten Abschnitt beschrieben wird.

10.1 Monaden

Eine Monade ist eine Struktur, die eine *Berechnung* beschreibt. Eine Berechnung gibt eventuell ein Resultat zurück, aber sie kann mehr als nur das Resultat sein. Z. B. kann eine Berechnung Seiteneffekte verursachen, Ausnahmen auslösen oder nichtdeterministisch zwischen mehreren Möglichkeiten auswählen. Was eine Berechnung genau ist, wird im Einzelfall entschieden.

Der Typ des Resultats einer Berechnung ist beliebig. Im allgemeinen wird der Typ einer Berechnung also durch einen Typkonstruktor M beschrieben, der noch mit dem Resultatstyp parametrisiert wird.

Eine Monade hat weiter zwei fundamentale Operationen. Eine binäre *Binde-Operation* ($\gg$) fügt Berechnungen zusammen. Ihr erstes Argument ist eine Berechnung vom Typ $\mathsf{M\ a}$, und ihr zweites Argument ist eine Funktion, die Werte vom Typ a auf Berechnungen vom Typ $\mathsf{M\ b}$ abbildet. Eine zusammengesetzte Berechnung

$$\mathsf{a} \gg \lambda \mathsf{x}.\ \mathsf{b}$$

wird dann informell gesagt so ausgewertet, daß zunächst a ausgeführt wird, das Resultat dann an x übergeben wird und mit b weitergefahren wird. Daneben gibt es in jedem Fall eine Operation unit, die ein Resultat zurückgibt. $\mathsf{unit\ x}$ ist die triviale Berechnung, die "gar nichts tut" und x zurückgibt. Was das alles genau heißt, kann in jedem Fall verschieden definiert werden. In jeder Monade müssen aber unit und $(\gg)$ bestimmte Gesetze erfüllen, die in der nächsten Definition aufgeführt werden.

Definition. Eine *Monade* ist ein Tripel $(\mathsf{M}, \gg, \mathsf{unit})$, wo

- M ein Typkonstruktor mit einem Elementtyp ist,
- $\gg$ und unit Operationen mit folgenden Signaturen sind:

$$\begin{array}{ll} (\gg) & : \mathsf{M\ a} \to (\mathsf{a} \to \mathsf{M\ b}) \to \mathsf{M\ b} \\ \mathsf{unit} & : \mathsf{a} \to \mathsf{M\ a}, \end{array}$$

- und folgende Gesetze gelten:

$$\begin{array}{ll} \mathsf{a} \gg (\lambda \mathsf{x.b} \gg \lambda \mathsf{y.c}) & = (\mathsf{a} \gg \lambda \mathsf{x.b}) \gg \lambda \mathsf{y.c} \\ \mathsf{unit\ x} \gg \lambda \mathsf{x.a} & = \mathsf{a} \\ \mathsf{a} \gg \lambda \mathsf{x.unit\ x} & = \mathsf{a}. \end{array}$$

Das heißt $(\gg)$ ist assoziativ mit (unit) als linker und rechter Identität.

Diese Definition legt fest, was allen Monaden gemeinsam ist. Einzelne Monaden legen fest, wie $(\gg)$ und (unit) implementiert werden, und haben daneben in der Regel noch weitere Operationen.

Zwei andere nützliche Operationen lassen sich mit unit und $(\gg)$ definieren.

$$\begin{array}{ll} \mathsf{skip} & : \mathsf{M\ ()} \\ (;;) & : \mathsf{M\ ()} \to \mathsf{M\ a} \to \mathsf{M\ a} \end{array}$$

$$\begin{array}{l} \mathsf{skip} = \mathsf{unit\ ()} \\ \mathsf{a\ ;;\ k} = \mathsf{a} \gg \lambda().\mathsf{k} \end{array}$$

Im folgenden werden wir annehmen, daß skip und $(;;)$ überall zusammen mit unit und $(\gg)$ definiert sind, auch wenn dies nicht immer explizit geschrieben wird.

Man bemerke, daß die Definition einer Monade M zwar eine Injektionsoperation unit umfaßt, die ein Element der Monade vom Typ $\mathsf{M\ a}$ liefert, aber keine Projektionsoperation, die es erlaubt, wieder aus der Monade herauszukommen — eine solche Projektionsoperation bräuchte den Typ $\mathsf{M\ a} \to \mathsf{a}$. Viele, wenn auch nicht alle Monaden lassen eine solche Projektionsoperation zu. Wenn die Projektion existiert, werden wir sie run nennen und mit der Monade mitdefinieren.

In den nächsten Abschnitten lernen wir verschiedene Monaden kennen.

10.2 Die Identitätsmonade

Die Identitätsmonade Id entsteht, wenn als Typkonstruktor und unit-Operation jeweils die Identität genommen wird und die $\gg$-Operation als Funktionsanwendung mit der Funktion als zweitem Argument aufgefaßt wird. Also:

$$\mathsf{type\ Id\ a} = \mathsf{a}$$

$$\begin{array}{ll} (\gg) & : \mathsf{Id\ a} \to (\mathsf{a} \to \mathsf{Id\ b}) \to \mathsf{Id\ b} \\ \mathsf{unit} & : \mathsf{a} \to \mathsf{Id\ a} \end{array}$$

$$\begin{array}{ll} \mathsf{a} \gg \mathsf{k} & = \mathsf{k\ a} \\ \mathsf{unit\ a} & = \mathsf{a} \end{array}$$

$$\begin{array}{ll} \mathsf{run} & : \mathsf{Id\ a} \to \mathsf{a} \\ \mathsf{run\ a} & = \mathsf{a} \end{array}$$

Eine Berechnung in der Identitätsmonade ist einfach ein Ausdruck, der ein Resultat berechnet.

10.3 Die Ausnahmemonade

Eine Berechnung in der Ausnahmemonade $(\mathsf{E\ a})$ ist ein Ausdruck, der entweder einen Fehlerwert $\mathsf{Err\ String}$ oder ein Resultat $\mathsf{Suc\ a}$ liefert. Die Monade wird folgendermaßen definiert:

```
type E a = Suc a | Err String

unit   : a → E a
(≫)    : E a → (a → E b) → E b

unit a  = Suc a
a ≫ k = case a of
          Suc x ⇒ k x
          Err s ⇒ Err s

run    : E a → a
run a  = case a of
          Suc x ⇒ x
          Err s ⇒ error "projection failed"
```

10.4 Die Listenmonade

Eine Berechnung in der Listenmonade liefert eine Liste von Resultaten anstelle eines einzigen Resultats. In der Listenmonade liefert unit a die Liste [a] mit a als einzigem Element. Wenn k vom Typ a → [b] ist, so bildet xs ≫ k die Funktion k auf alle Elemente der Liste xs ab — dies liefert also eine Liste vom Typ [[b]] — und konkateniert anschließend alle Teillisten, was eine Liste vom Typ [b] ergibt. Die genaue Definition ist:

```
type L a = [a]

unit   : a → L a
(≫)    : L a → (a → L b) → L b

unit a  = [a]
a ≫ k = concat (map k a)

run    : L a → a
run a  = head a
```

10.5 Die Arraytransformer-Monade

Wir beschreiben nun, wie Monaden Zustand ausdrücken und kapseln können. Wir fangen mit einer besonders einfachen Version von Zustand

an, nämlich einem einzigen eindimensionalen Array. Als Operationen auf Arrays nehmen wir an:

```
type Vec a = Array Int a

emptyarray : Vec a
append     : Vec a → a → Vec a
update     : Vec a → Int → a → Vec a
lookup     : Vec a → Int → a
length     : Int
```

emptyarray liefert einen Array der Länge 0. append hängt ein Element an einen Array an. Das Resultat ist ein neuer Array, der mit dem Argument keinen Speicher teilt. Es muß also kopiert werden. update ändert ein Element eines Arrays. Das Resultat ist wieder ein neuer Array, der mit dem Argument keinen Speicher teilt. Ebenso wie append ist also update eine sehr teure Operation. lookup liest ein Element des Arrays. Schließlich gibt length die Anzahl der Elemente eines Arrays an.

Im folgenden zeigen wir, wie Arraymanipulationen mit der Effizienz von imperativen Sprachen ausgeführt werden können. Der Einfachheit halber betrachten wir nur eindimensionale Arrays und fixieren den Elementtyp auf z. B. Float. Eine Berechnung (vom Typ AT a) in der Arraytransformer-Monade ist dann eine Funktion, die einen Array als Argument enthält und einen veränderten Array und dazu noch ein Resultat (vom Typ a) zurückgibt. Hier ist die volle Definition:

```
type AT a = Vec Float → (Vec Float, a)
```

Monaden-Operationen:

```
unit   : a → AT a
(≫)    : AT a → (a → AT b) → AT b

unit a  = λv.(v, a)
a ≫ k = λv.let (v', x) = a v in k x v'

run    : AT a → a
run a  = let (v, x) = a emptyarray
         in x
```

Operationen auf Arrays:

```
upd    : Int → Float → AT ()
app    : Float → AT ()
rd     : Int → AT Float
elems  : AT ([Float])

upd i x = λv. (update v i x, ())
app x   = λv. (append v x, ())
rd i    = λv. (v, lookup v i)
elems   = λv. (v, [lookup v i | i ← 0 .. length v − 1])
```

Hier ist ein Beispiel, das die Arraytransformermonade zur Anfertigung eines Histogramms benutzt. Die Eingabe der Funktion histogram besteht aus einer Zahl l und einer Liste von Paaren (i, x), wobei i ein Index in $\{0, \ldots, l-1\}$ ist und x ein Datenwert vom Typ Float ist. Die Funktion liefert als Resultat eine Liste, die für jeden Index die Summe aller Werte auf diesem Index angibt.

```
histogram : Int → [(Int, Float)] → [Float]

histogram l xs =
  run (initialize l ;; accumulate xs)

initialize l =
  if l == 0 then skip
  else app 0.0 ;; initialize (l−1)

accumulate [] =
  elems
accumulate ((i,x):xs) =
  rd i ≫ λy.upd i (x + y) ;; accumulate xs
```

Wie sieht es mit der Effizienz dieses Programms aus? Die Funktion accumulate führt für jedes Element der Argumentliste eine upd-Operation aus, die wiederum mit Hilfe von update implementiert ist. Die Gesamtkosten liegen also bei $O(mn)$, wo m die Länge der Eingabeliste und n die Obergrenze der Indizes ist. Das liegt um einen Faktor n höher als eine imperative Implementierung des Problems.

Diese Argumentation setzt voraus, daß update durch Kopieren implementiert wird. Dies muß aber nicht so sein! Man beachte, daß das Histogramm-

Programm Arrays nie direkt verwendet, sondern nur die Operationen aufruft, die in der Arraytransformer-Monade definiert wurden. Wenn wir uns diese Operationen anschauen, stellen wir fest, daß jede Operation einen Array als Argument übernimmt und einen Array als Resultat liefert. Solche Operationen heißen auch *linear* (im Arraytyp). Es folgt, daß zu jedem Zeitpunkt der Abarbeitung des Histogramm-Programms nur genau ein Array referenziert wird — alle früheren Versionen des Arrays werden zur Berechnung nicht mehr benötigt. Aber das heißt, daß wir in Wirklichkeit nur *eine* Kopie des Arrays benötigen, auf der alle Modifikationen destruktiv, also durch Überschreiben, ausgeführt werden können!

Es bleibt noch die Frage, wie ein Übersetzer funktionaler Programme diese Situation erkennen soll, um diese Optimierung durchzuführen. Eine Möglichkeit bieten Datenflußanalysen [16, 3] oder spezielle Typsysteme [21, 17]. Diese haben den Vorteil, daß sie allgemein einsetzbar sind, aber den Nachteil, daß sie oft sehr aufwendig und für den Programmierer undurchsichtig sind.

Eine andere Möglichkeit ist, eine Bibliothek von abstrakten Datentypen zu schaffen, die alle destruktiv implementiert werden können. Die abstrakte Version der Arraytransformer-Monade wäre einfach:

```
type AT a;

unit  : a → AT a
(≫)   : a → (a → AT b) → AT b

upd   : Int → Float → AT ()
app   : Float → AT ()
rd    : Int → AT Float
elems : AT ([Float])
```

Sowohl die Definition des Typ von Arraytransformern AT a als auch auch die Implementierungen der Operationen, die auf Zustand zugreifen, sind versteckt worden, und der Zustandstyp Vec Float taucht gar nicht mehr auf.

Ein Programm, das diesen abstrakten Datentyp benutzt, kann auf den Array also nur noch mit den exportierten Operationen zugreifen, die alle linear im Arraytyp sind. Es folgt, daß Änderungen des Arrays destruktiv implementiert werden können. (Eine solche Implementierung kann natürlich nicht in der rein funktionalen Sprache selbst ausgeführt werden.)

10.6 Zustandstransformer

Der letzte Abschnitt hat gezeigt, wie Zugriffe auf eine einzige "Zustands-Struktur" mit der Effizienz imperativer Programme ausgeführt werden können. In diesem Abschnitt zeigen wir, wie dies auf alle möglichen Seiteneffekte ausgeweitet werden kann. In der Monade IO kann eine Berechnung beliebige Speicherzugriffe und Ein-Ausgabeoperationen auslösen. Die Monade ist wie folgt definiert:

```
type IO a;
type Ref a;

unit    : a → IO a
(≫)     : IO a → (a → IO b) → IO b

newref  : IO (Ref a)
rd      : Ref a → IO a
(:=)    : Ref a → a → IO ()

syscall : String → [Int] → IO Int
```

Hier alloziert newref eine neue Variable; rd liest die Variable aus, und (:=) weist ihr einen neuen Wert zu. syscall name args führt eine Operation mit Namen name und Argumenten args aus. Die Operation kann z. B. eine imperative Bibliotheks-Prozedur sein. Der Einfachheit halber wird angenommen, daß Argumente und Resultat Ganzzahlen sind.

Die run-Operation fehlt diesmal, da es im Allgemeinen nicht möglich ist, durch Ein- und Ausgabe verursachte Seiteneffekte wieder rückgängig zu machen, eine Operation vom Typ IO a → a also semantisch nicht möglich ist. Wenn Seiteneffekte nur auf den Speicher beschränkt sind, ist dagegen eine Kapselung mittels run möglich, wenn das Hindley/Milner Typsystem leicht erweitert wird. Siehe dazu [11].

Andere imperative Sprachkonstrukte wie z. B. Schleifen lassen sich aufbauend auf diesen Operationen so definieren:

```
while : IO Bool → IO () → IO ()
while cond stat =
  cond ≫ λc.
  if c then stat ;; while cond stat
  else unit ()
```


Hier ist ein Beispiel eines Programms, das die IO-Monade benutzt:

```
echo: IO ()

getc   = syscall "getc" []
putc x = syscall "putc" [x]

echo =
  getc ≫ λx.
  if x == eofch then unit ()
  else putc x ;; echo
```

Eine interessante Frage ist, ob solche Programme imperativ oder funktional sind. Sie sind wohl beides — imperativ, weil sich alle imperativen Programmkonstrukte darstellen lassen, und funktional, weil dies explizit erfolgt. Imperative und funktionale Teile können durch ihren Typ auseinandergehalten werden. Alle Gesetze des funktionalen Programmierens gelten nach wie vor.

10.7 Ein Beispiel

Der Programmierer kann sich natürlich auch seine eigenen Monaden definieren, was oft zu einer besseren Strukturierung des Programms führt. Als Beispiel kommen wir auf das anfangs dieses Kapitels beschriebene Typrekonstruktionsprogramm zurück. Wir strukturieren das Programm neu, indem wir alle globalen Variablen in einer Transformermonade TP a verstecken. Die Neufassung des Programms ist wie folgt.

Typen:

```
type TVar  = Int
type Var   = Int
type Type  = TV TVar | Type ≻ Type
type Term  = Id Var | Lam Var Term | App Term Term
type Subst = TVar → Type
type E a   = Suc a | Err String
type Typo  = [(TVar, Type)]

type TP a  = (Int, Subst) → E (Int, Subst, a)
```

Monaden-Operationen:

```
unit : a → TP a
(≫) : TP a → (a → TP b) → TP b

unit a =
  λ(n, s). Suc (n, s, a)

a ≫ k =
  λ(n, s).
    case a (n, s) of
      Suc (n', s', x) ⇒ k x (n', s')
      Err s ⇒ Err s
```

Operationen auf Substitutionen und Typvariablen:

```
newtvar : TP TVar
newtvar =
  λ(n, s). Suc (n+1, s, n)

deref : Type → TP Type
deref (TV a) =
  λ(n, s).
    case s a of
      TV b where a /= b ⇒ deref (TV b) (n, s)
deref t =
  λ(n, s). t

assign : TVar → Type → TP ()
assign a t =
  λ(n, s).
    Suc (n, λb. if a == b then t else s b, ())
```

Operationen auf Typothesen:

```
insert : Typo → (Var, Type) → Typo
insert T (x, t) = (x, t) : T

lookup : Typo → Var → TP Type
lookup [] y =
  λ(n, s). Err "undeclared variable"
lookup ((x,t):xs) y =
```



```
  λ(n, s).
    if x == y then Suc (n, s, t)
    else lookup xs y
```

Unifikation:

```
occursCheck: TVar → Type → TP ()
occursCheck a t =
  deref t ≫ λt'.
  case t' of
    TV b ⇒
      if a == b then Err "recursive type"
      else skip
    t₁ ≻ t₂ ⇒
      occursCheck a t₁ ;; occursCheck a t₂

unify : (Type, Type) → TP ()
unify (t₁, t₂) =
  deref t₁ ≫ λt₁'.
  deref t₂ ≫ λt₂'.
  case (t₁', t₂') of
    (TV a, TV b) where a == b ⇒
      unit ()
    (TV a, t), (t, TV a) ⇒
      occursCheck a t ;; assign a t
    (s₁ ≻ t₁, s₂ ≻ t₂) ⇒
      unify s₁ s₂ ;; unify t₁ t₂
    _ ⇒
      Err "types incompatible"
```

Typ-Rekonstruktion:

```
tp : (Typo, Term, Type) → TP ()

tp (T, Id x, t) =
  lookup T x ≫ λt'.
  unify (t, t')

tp (T, Lam x e, t) =
  newtvar ≫ λa.
  newtvar ≫ λb.
  unify (a ≻ b, t) ;;
```

```
    tp (insert T (x, a), e, b)

  tp (T, App d e, t) =
    newtvar ≫ λa.
    tp (T, d, a ≻ t) ;;
    tp (T, e, a)
```

Im Vergleich zu der ersten Lösung sind die Zugriffsoperationen teilweise komplexer, aber die Implementierung der Hauptfunktion tp hat sich erheblich vereinfacht. Der Gesamtgewinn an Lesbarkeit würde bei einem größeren Problem noch deutlicher. Die Monadenstruktur kann außerdem verwendet werden, um die Effizienz des Programms zu steigern.

Aufgabe 10.2 Implementieren Sie den Algorithmus möglichst effizient, unter Zuhilfenahme imperativer Programmierkonstrukte. Achten Sie dabei darauf, daß Sie nur die Monaden-Operationen ändern, nicht jedoch die Funktionen *tp* und *unify*.

Teil III

Prozesse

Dieser Teil ist noch nicht so detailliert ausgearbeitet, wie die vorhergehenden Teile. Das Studium dieses Teiles sollte umfassen:

- Das Skript, soweit es vorhanden ist,
- Folienkopien zum Objekt-Orientierten Programmieren und zum Parallelen Programmieren,
- Das Papier *Applying Pi: Towards a Basis for Concurrent Imperative Programming.*

Kapitel 11

Grundlagen reaktiver Programme

Im letzten Teil des Skripts haben wir Funktionen als Grundlage von sequentiellen, deterministischen Proggrammen studiert. Wir haben gesehen, daß auch viele andere Programmkonstrukte als Funktionen darstellbar sind, z. B. wurde eine imperative Anweisung als Funktion von einem Eingangszustand in einen neuen Zustand und einen Resultatsteil aufgefaßt. In diesem Sinn sind Funktionen die Grundlage aller algorithmischen Sprachen, die Eingabe auf deterministische Weise in Ausgabe umformen,

Nicht alle Programme lassen sich aber auf natürliche Weise als Algorithmen beschreiben. *Reaktive* Programme lassen sich nur sehr schwer in ein algorithmisches Schema presse. Typische Eigenschaften solcher Programme sind:

- Die wiederholte Interaktion mit einer Umgebung, statt des einfachen Ein/Ausgabeschemas eines Algorithmus.
- Die Abhängigkeit von einem Zustand, der sich mit der Zeit ändert, und der das Verhalten des Programms bestimmt.
- Der parallele Ablauf mehrerer logischer Berechnungsabläufe, im Programm selbst (Prozesse) oder in der Umgebung (asynchrone Ereignisse). Dieser Aspekt wird auch *Nebenläufigkeit* genannt
- Ein gewisser Grad an Nichtdeterminismus, der in der Regel schon durch Nebenläufigkeit bedingt wird.

Reaktive Programme und nebenläufige Programme treten in vielen Varianten auf. Beispiele sind:

- Ein Betriebssystem, das meherere Prozesse verwaltet.
- Ein Programm, das Ein/Ausgabe-Operationen mit Berechnungen verschränkt.
- Eine graphische Schnittstelle, die Eingabeaktionen des Benutzers und Ausgabeaktionen verschiedener Prozesse koordiniert.
- Ein Programm, das Alarme (wie z. B. den von Control-C ausgelösten) behandelt.
- Ein Router für Nachrichtenpakete in einem Netz.

Die beiden wichtigsten Sprachparadigmen für reaktive Programme sind das *objekt-orientierte* Programmieren, das es erlaubt den Zustand eines Programms zu modellieren und zu modularisieren, und das *parallele Programmieren*, das es erlaubt, nebenläufige Prozesse auszudrücken.

Sowohl Zustandsbegriff wie auch Nebenläuifigkeit ist allerdings auch ohne die anderen Eigenschaften reaktiver Programme möglich. Beispiele sind:

- Ein funktionales Programm, das Ergebnisse von Zwischenberechnungen zur Effizienzsteigerung in einer mutierbaren Datenstruktur festhält.
- Ein Programm, das zur Effizienzsteigerung auf mehreren Prozessoren in einem Computer oder auf mehreren Computern in einem Netz läuft.
- Ein Programm, das in meherere Prozesse zerlegt wird, um Speicherlatenzzeiten zu verdecken (während ein Prozeß auf den Haupt- speicher zugreift, kann ein anderer rechnen $\rightarrow$ MultithreadArchitektur)

Solche Programme können auf hoher Ebene als Algorithmen verstanden werden, und z. B. funktional ausgedrückt werden. Nebenläufigkeit beschreibt hier lediglich die Implementierungsaspekte.

Reaktive und nebenläufige Programme sind schon sehr weitverbreitet, und dies wird aller Wahrscheinlichkeit nach in Zukunft noch zunehmen, einmal wegen der fortschreitenden Vernetzung, und auch weil parallele Hardware mit höherer Schaltkreis-Integration immer kostengünstiger wird. Welche Hilfsmittel gibt es nun, um solche Programme zu formalisieren und zu verstehen?

Als erstes sollten wir versuchen das anzuwenden, was wir schon studiert haben, in unserem Fall das λ-Kalkül. Da Reduktion im λ-Kalkül überall in einem Term angewandt werden kann, ist es vorstellbar, daß man Nebenläufigkeit durch parallele Reduktion beschreiben kann. Allerdings sagt uns der Church-Rosser Satz, daß alle Reduktionsfolgen immer dasselbe Resultat (d. H. dieselbe schwache Normalform) liefern. Das λ-Kalkül kann also sicher nur für eine eingeschränkte Klasse nebenläufiger Programme verwendet werden, in der alle möglichen Ausführungspfade dasselbe Resultat liefern.

Um auch andere Programme beschreiben zu können, benötigen wir eine neue Grundlage, die im nächsten Kapitel vorgestellt wird.

11.1 Ein Prozeß-Kalkül

Warum ist der Lambda Kalkül deterministisch? Eine mögliche Intuition ist, daß Argumente und Funktionen eindeutig einander zugeordnet sind, verschiedene Reduktionsfolgen sich also nicht gegenseitig stören können. Die Zuordnung geschieht dabei über die *Position* von Funktion und Argument.

Um Nichtdeterminismus gezielt einführen zu können, geben wir die Zuordnung von Funktion und Argument über deren Position auf, und ersetzen sie durch eine Zuordung über einen *Namen*. Anstelle von $(\lambda x.d)e$ schreiben wir jetzt

$$\nu a.(a^? x.P \mid aQ)$$

Dabei ist a der Name eines *Kanals*, über den eine *Interaktion* abläuft. Neue Namen können durch den ν-Binder geschaffen werden. $a^? x.P$ ist ein *Eingabeterm*, der ein Datum vom Kanal a liest, dies für die Variable x einsetzt, und danach mit P weiterfährt. aQ ist ein *Ausgabeterm*, der das Resultat der Berechnung von Q an den Kanal a sendet. Der $(|)$ Operator drückt *parallele Komposition* aus; er ist assoziativ und kommutativ, so daß die Reihenfolge der Terme P_i in $P_1 \mid \ldots \mid P_n$ keine Rolle spielt.

Reduktion in diesem Kalkül ist ähnlich wie Reduktion im Lambda-Kalkül, aber einfacher, da nur Namen, nicht aber ganze Terme oder Werte, übergeben werden können:

$$a^? x.P \mid ab \quad \rightarrow \quad [b/x]P$$

Man beachte die Analogie zur β-Reduktion. Der Prozeßkalkül geht aber über den funktionalen Kalkül hinaus, indem nichtdeterministische Auswahl zwischen mehreren Interaktionen möglich ist, wie etwa in dem Fall

$$a^?x.P \mid a^?y.Q \mid ab.$$

Dieser Term läßt sich sowohl zu $[b/x]P$ als auch zu $[b/y]Q$ reduzieren, und die beiden reduzierten Terme lassen sicht im allgemeinen nicht mehr zusammenführen, d. H. Reduktion ist nicht konfluent.

Dieser einfache Kalkül heißt *angewandter π-Kalkül*. Er soll im folgenden genauer studiert werden.

Abbildung 11.1 beschreibt die Syntax von π_a. Wir gehen aus von einem Alphabet von *Namen*, die mit Kleinbuchstaben geschrieben werden. Ein *Term* (oder *Agent*) in π_a setzt sich aus folgenden Konstrukten zusammen.

- Der *Ausgabeterm* aP. Hier ist a ein Name und P ein Term.
- Der *Eingabeterm* $a^?x.P$. Die Variable x wird in diesem Term gebunden.
- Die parallele Komposition zweier Terme P und Q, $P \mid Q$.
- Die *Namensrestriktion* $\nu a.P$. Dies schränkt den Sichtbarkeitsbereich des Namens a auf P ein. Operationell entspricht dieser Term dem Anlegen eines neuen Namens für a.
- Die *Replikation* $!P$. Dies führt beliebig viele Kopien von P parallel aus. Replikation ersetzt die Rekursion in funktionalen Sprachen. [1].
- Die *Identität* 0. Dies bezeichnet einen Agenten, der gar nichts tut.
- Der *Wächter* $[a = b]P$. Dies führt den Term P aus, wenn $a = b$ und blockiert sonst.
- Das *Resultat* a. Resultat eines Agenten kann also ein Name sein.

Wir benutzen die Buchstaben P, Q, R für Agenten.

Wir kennen bereits aus dem Lambda-Kalkül, daß wir von der exakten Term-Syntax etwas abstrahieren, und immer mit strukturellen Äquivalenzklassen von Termen arbeiten. Im Lambda-Kalkül sind genau die Terme strukturell äquivalent, die ineinander α-umbenennbar sind. In π_a sind solche Terme auch strukturell äquivalent, aber nicht nur sie. Strukturelle Äquivalenz ($\equiv$)

[1] Rekursion ist in π_a ohne (!) nicht modellierbar, da nur Namen übergeben werden, Terme also nie kopiert werden

Abstrakte Syntax

Namen	$a, b, c, ..., x, y, z$		
Termes	P, Q, R	$= x$	Name
		$\mid x^?y.M$	Abstraction(Eingabe)
		$\mid xM$	Application (Ausgabe)
		$\mid M \mid N$	Parallele Komposition
		$\mid \nu x.M$	Restriktion
		$\mid !M$	Replikation
		$\mid \mathsf{0}$	Identität
		$\mid [a = b]M$	Wächter

Strukturelle Äquivalenz ($\equiv$)

(α)	$a^?x.P$	$\equiv$	$a^?y.[y/x]P$	$(y \notin \mathrm{fv}(P))$
	$\nu a.P$	$\equiv$	$\nu b.[b/a]P$	$(b \notin \mathrm{fc}(P))$
($\mid$)	$(P \mid Q) \mid R$	$\equiv$	$P \mid (Q \mid R)$	
	$P \mid Q$	$\equiv$	$Q \mid P$	
	$P \mid 0$	$\equiv$	P	
(ν)	$P \mid \nu a.Q$	$\equiv$	$\nu a.(P \mid Q)$	$(a \notin fv(P))$
	$\nu a.0$	$\equiv$	0	
(!)	$!P$	$\equiv$	$P \mid !P$	
(*apply*)	$a(P \mid Q)$	$\equiv$	$aP \mid aQ$	
	$a(b^?x.P)$	$\equiv$	$b^?x.P$	
	$a(\nu b.P)$	$\equiv$	$\nu b.aP$	$(a \neq b)$
(*match*)	$[a = a]P$	$\equiv$	P	

Reduktion

$$(com) \qquad a^?x.P \mid ab \;\longrightarrow\; [b/x]P$$

Abbildung 11.1: Der angewandte π-Kalkül, π_a

ist in π_a die kleinste Äquivalenzrelation, die alle ($\equiv$)-Regeln aus Abbildung 11.1 enthält.

Reduktion eines Agenten wird durch folgende Interaktions-Regel beschrieben.

$$(com): \qquad a^?x.P \mid ab \quad \rightarrow \quad [b/x]p$$

Reduktion kann überall stattfinden, *außer* in einem Eingabeterm oder in einer Replikation. Das heißt, zur vollständigen Definition von $\rightarrow$ mit einem Ableitungssystem kommen noch die folgenden zwei Regeln hinzu:

$$\frac{P \rightarrow P'}{\nu a.P \rightarrow \nu a.P'} \qquad\qquad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}.$$

Beispiel 11.1 1. Ein Puffer

$$FWD(a,b) \quad \stackrel{\text{def}}{=} \quad !a^?x.bx$$

der alle Nachrichten, die an a gesendet werden, an b weiterleitet.

2. Ein Gleichheitstester

$$EQ?(a,b,P) \quad \stackrel{\text{def}}{=} \quad \nu c.(ac \mid b^?x.P)$$

Dieser Tester ist äquivalent zum Wächter $[a = b]P$, vorausgesetzt es fließen keine anderen Nachrichten über x und y.

3. Ein bidirektionaler Puffer

$$\begin{aligned} EQ(a,b) \quad &\stackrel{\text{def}}{=} \quad FWD(a,b) \mid FWD(b,a) \\ &\stackrel{\text{def}}{=} \quad !a^?x.bx \mid !b^?x.ax. \end{aligned}$$

Ein solcher Puffer beeinflußt den Gleichheitstester $EQ?(a,b,P)$:

$$EQ(a,b) \mid EQ?(a,b,P) \quad \twoheadrightarrow \quad P.$$

Der Wächter $[a = b]P$ bleibt hingegen unbeeinflußt. Wenn $a \neq b$, so gilt

$$EQ(a,b) \mid [a = b]P \quad \not\twoheadrightarrow \quad P.$$

Beispiel 11.2 Wir modellieren nun ein mobiles Telephonsystem in π_a. Das System wird gebildet von einem Auto, das sich zwischen zwei Übertragungs-Basen bewegt. Je nach Standort des Autos ist eine Basis inaktiv während die andere Gespräche überträgt. Das Auto ist mit der aktiven Basis über zwei Gesprächskanäle und zwei Signalkanäle (je einen in jede Richtung) verbunden. Das Verlassen eines Sendebereichs einer Base zeigt das Auto durch Meldung auf dem Signalkanal (outsignal Weak) an. Danach muß die jeweils inaktive Basis aktiviert werden, und alle Kanäle umgeleitet werden. Dies wird durch folgendes Prozeß-System ausgedrückt.


```
CAR (outTalk, inTalk, outSignal, inSignal) =
  ( outTalk Blabla
  | outSignal Weak)
   |
  ( inTalk? blabla. CAR (outTalk, inTalk, outSignal, inSignal)
  | inSignal? cs'. CAR cs')

BASE (outOther, inOther) (outTalk, inTalk, outSignal, inSignal) =
  ( outTalk? blabla. (RelayOut blabla |
         BASE (outOther, inOther) (outTalk, inTalk, outSignal, inSignal))
  | Relay?.in blabla. (c.inTalk blabla |
        BASE (outOther, inOther) (outTalk, inTalk, outSignal, inSignal))
  | outSignal? weak. outOther ()
  | inOther (outTalk', inTalk', outSignal', inSignal').
        (inSignal (outTalk', inTalk', outSignal', inSignal') |
         IDLEBASE (outOther, inOther))

IDLEBASE (outOther, inOther) =
  inOther ().
    νTi.νTo.νSi.ν So.
        outOther(To, Ti, So, Si) |
        BASE (outOther, inOther) (To, Ti, So, Si)
```

11.2 Kodierung von sequentiellen Programmierkonstrukten

Siehe dazu das Papier *Applying Pi: Towards a Basis for Concurrent Imperative Programming.*

11.3 Variation: Der asynchrone π-Kalkül

Der angewandte π-Kalkül läßt sich noch etwas weiter vereinfachen, wenn wir nur Namen als Argumente von Ausgabetermen vorsehen. Das heißt, ein Term wie

```
a (b c)
```

wäre unzulässig. Statt dessen müßte man z. B.

$$\nu r.r^? x.ax \mid b(c,r)$$

schreiben. Hier ist r ein neueingeführter *Resultatskanal*, der dem Kanal b als zusätzliches Argument geschickt wird.

Daß eine solche Kodierung immer möglich ist, die Einschränkung also die Ausdrucksfähigkeit nicht grundsätzlich einschränkt, wird im obenerwähnten Papier *Applying Pi: Towards a Basis for Concurrent Imperative Programming* gezeigt.

11.4 Variation: Der π-Kalkül mit Sende-Synchronisierung

Der π-Kalkül mit Sende-Synchronisierung [12, 13] entsteht aus dem asynchronen Kalkül, wenn man die Sende-Operation um einen Prozeßteil ergänzt, der ausgeführt wird, wenn die gesendete Nachricht empfangen wird.

$$P \quad ::= \quad \ldots \mid ab.P \qquad \text{Ausgabe}$$

Anders als beim asynchronen Kalkül kann also auf Senderseite auf den Empfang einer Nachricht gewartet werden. Oft wird auch noch eine nicht-deterministische Auswahloperation $P + Q$ dazugenommen. Reduktion in diesem Kalkül wird dann folgendermaßen definiert:

$$(ab.P + \ldots) \mid (a^? x.Q + \ldots) \quad \rightarrow \quad P \mid [b/x]Q$$

Ursprünglich waren Kalküle mit synchroner Kommunikation (und deren Implementierungen) wesentlich populärer als solche mit asynchroner Kommunikation, was hauptsächlich damit zusammenhing, daß sich synchrone Kommunikation ohne zusätzlichen Puffer-Speicher durchführen läßt, während asynchrone Kommunikation je nach Ausführungsreihenfolge im Prinzip unbeschränkt viel Pufferspeicher benötigt.

Auch der ursprüngliche π-Kalkül [13] hatte synchrone Kommunikation. Wir konzentrieren uns hier auf die asynchrone Variante, weil sie wesentlich einfacher zu handhaben ist. Außerdem sind Implementierungen nebenläufiger Prozesse ebenfalls zunehmend asynchron, um die Verzögerung, die durch einen vollständiger Nachrichtenaustausch mit Antwort bedingt wird, zu vermeiden ("Latenzzeitverdeckung" [20]).

11.5 Prozeß-Verfeinerung und Äquivalenz

Bisher haben wir den Prozeßkalkül nur als abstrakte Maschine kennengelernt. Neben diesem operationellem Verständnis sollte aber auch eine abstraktere Sicht treten, in der erklärt wird, wann ein Programm ein anderes verfeinert, bzw äquivalent zu einem anderen ist. Dies soll nun am Beispiel des asynchronen π-Kalküls geschehen.

Im λ-Kalkül war hatten wir die observationelle Äquivalenz $\cong$, deren Definition hier zur Erinnerung noch einmal wiederholt ist.

Definition. Ein Term e *konvergiert*, in Symbolen $e \Downarrow$, wenn $e \twoheadrightarrow v$ für einen beliebigen Wert v.

Definition. Zwei Terme d, e sind *operationell äquivalent*, in Symbolen $d \cong e$, wenn für jeden Kontext C, so daß $C[d]$ und $C[e]$ abgeschlossen sind, gilt:

$$C[d] \Downarrow \quad \text{genau dann wenn} \quad C[e] \Downarrow .$$

Wenn wir diese Definition übernehmen wollen, stellt sich die Frage, was $\Downarrow$ in einem nebenläufigen, nicht-deterministischen System bedeuten soll. Zunächst müssen wir ein Analogon zu dem funktionalen "Wertbegriff" finden. Da Werte im funktionalen Programmieren λ-Abstraktionen sind liegt es nahe, statt "Wert" irgendeinen Prozeß zu nehmen, der mit seiner Umgebung kommunizieren kann.

Definition. Die *aktiven Namen* $\mathcal{AN}(P)$ eines Prozesses P sind rekursiv wie folgt definiert.

$$\begin{array}{lcl}
\mathcal{AN}(ab) & = & \{a\} \\
\mathcal{AN}(a^? x.P) & = & \{a\} \\
\mathcal{AN}(P \mid Q) & = & \mathcal{AN}(P) \mid \mathcal{AN}(Q) \\
\mathcal{AN}(\mathsf{0}) & = & \emptyset \\
\mathcal{AN}(\nu a.P) & = & \mathcal{AN}(P) \backslash \{a\} \\
\mathcal{AN}(!P) & = & \mathcal{AN}(P)
\end{array}$$

Ein Prozeß heißt *aktiv* wenn $\mathcal{AN}(P) \neq \emptyset$, andernfalls heißt er *inaktiv*.

Jetzt sind immer noch zwei Möglichkeiten zur Interpretation von $\Downarrow$ denkbar.

1. $P \Downarrow$, wenn es einen aktiven Prozeß Q gibt, so daß $P \twoheadrightarrow Q$.
2. $P \Downarrow$, wenn jede von P ausgehende Reduktionsfolge in einem aktiven Term endet.

Die erste Interpretation wird auch "Kann-Konvergenz" genannt; in Symbolen: $P \Downarrow_?$. Die zweite Interpretation wird auch "Muß-Konvergenz"; in Symbolen: $P \Downarrow_!$. Die beiden Arten von Konvergenz erzeugen jeweils eine operationelle Äquivalenzrelation.

Definition. Sei $X \in \{?, !\}$. Zwei Terme P, Q sind *operationell äquivalent* bzgl. der Konvergenz $\Downarrow_X$, in Symbolen $d \cong_X e$, wenn für jeden Kontext C, so daß $C[d]$ und $C[e]$ abgeschlossen sind, gilt:

$$C[d] \Downarrow_X \quad \text{genau dann wenn} \quad C[e] \Downarrow_X .$$

Daß $\cong_?$ und $\cong_!$ zwei verschiedene Äquivalenzrelationen sind, zeigt das folgende Beispiel.

Beispiel 11.3 Sei $\Omega = \nu a.(!a^?x.\mathsf{0} \mid !ab)$ ein inaktiver Prozeß, der nicht terminiert. Dann gilt:

$$\Omega \mid bc \;\cong_?\; bc \qquad \text{aber} \qquad \Omega \mid bc \;\ncong_!\; bc.$$

Umgekehrt gilt:

$$\Omega \mid bc \;\ncong_?\; \Omega \qquad \text{aber} \qquad \Omega \mid bc \;\cong_!\; \Omega.$$

Aufgabe 11.1 (a) Finden Sie einen Kontext C, so daß $C[\Omega \mid bc] \Downarrow_!$ aber nicht $C[bc] \Downarrow_!$.
(b) Finden Sie einen Kontext C, so daß $C[\Omega \mid bc] \Downarrow_?$ aber nicht $C[\Omega] \Downarrow_?$.

Beispiel 11.3 hat gezeigt, daß $\cong_?$ und $\cong_!$ unvergleichbar sind. Wesentlich dabei ist, daß Reduktion sich auf den nicht-terminierenden Prozeß Ω beschränken kann, und so jede Interaktion mit der Umgebung verhindern kann. Diese Art der Ausführung wird auch *unfair* genannt. Demgegenüber sind auch *faire* Ausführungsstrategien möglich:

1. Wenn in einer undendlichen Reduktionsfolge zu einem bestimmten Zeitpunkt eine Kommunikation über einen Kanal a stattfinden kann, so wird eine Kommunikation über den Kanal a in einem darauffolgenden Zweitpunkt sicher stattfinden (Kanalfairness)

2. In einer unendlichen Reduktionsfolge kann ein Unterterm nicht unendlich oft an einer Reduktion teilnehmen, ohne tatsächlich reduziert zu werden (Prozeßfairness).

Aufgabe 11.2 Welche der Äquivalenzen in Beispiel 11.3 ändern sich, wenn (Prozeß-)Fairness vorausgesetzt wird?

Die besprochenen Äquivalenzen haben allerdings alle den Nachteil, daß sie sehr schwer nachzuweisen sind, da sie eine Berücksichtigung aller möglichen Programmtexte erfordern. Es ist im Allgemeinen günstiger, schrittweise vorzugehen, und jeweils nur die im nächsten "Zug" möglichen Aktionen zweier Prozesse zu vergleichen. Dies führt dann zum Begriff der *Simulation*. Simulation erklärt sich über die *Aktionen* eines Prozesses.

Definition. Eine *Aktion* α ist ein Element aus $\{ab, a(\nu b), a^?b\}$. ab und (νb) sind *Ausgabeaktionen*; $a^?b$ ist eine *Eingabeaktion*. Wir schreiben $P \stackrel{ab}{\Longrightarrow} P'$, wenn $P \twoheadrightarrow ab \mid P'$. Wir schreiben $P \stackrel{a(\nu b)}{\Longrightarrow} P'$, wenn $P \twoheadrightarrow \nu b.(ab \mid P')$. Schließlich schreiben wir $P \stackrel{a^?b}{\Longrightarrow} P'$, wenn $P \mid ab \twoheadrightarrow P'$.

Definition. Eine Relation $\sim$ zwischen Prozessen ist eine *Simulation*, wenn für jedes Paar von Prozessen P, Q und für jede Aktion α gilt:

> Wenn $P \sim Q$ und $P \stackrel{\alpha}{\Longrightarrow} P'$ so gibt es ein Q', so daß $Q \stackrel{\alpha}{\Longrightarrow} Q'$ und $P' \sim Q'$

Das heißt, P simuliert Q, wenn jede mögliche Aktion von P auch eine mögliche Ausgabeaktion von Q ist, und für die Rest-Prozesse P' und Q' wieder gilt, daß P' Q' simuliert. Anders ausgedrückt ist der Prozeß P eine deterministischere Fassung des Prozesses Q.

Äquivalenz von Prozessen wird durch wechselseitige Simulation erklärt:

Definition. Geben eine Relation $\sim$, sei $\sim^{-1}$ die Relation, für die gilt:

$$Q \sim^{-1} P \qquad \Leftrightarrow \qquad P \sim Q.$$

Definition. Eine Relation $\sim$ ist eine *Bisimulation*, wenn $\sim$ und $\sim^{-1}$ Simulationen sind.

Definition. Zwei Prozesse P und Q heißen *bi-ähnlich*, in Symbolen $P \approx Q$ wenn es eine Bisimulation $\sim$ gibt, so daß $P \sim Q$ gilt.

Satz 11.4 $\approx$ ist eine Bisimulation.

Die Definition von Bi-Ähnlichkeit liefert direkt ein mächtiges Beweisprinzip. Um nämlich zu zeigen, daß zwei Prozesse bi-ähnlich sind ($P \approx Q$), genügt es, irgendeine Bisimulation $\sim$ zu finden, so daß $P \sim Q$.

Beispiel 11.5 Wir zeigen, daß für beliebiges P gilt: $\mathsf{0} \approx \nu a.a^? x.P$.

Betrachte die Relation

$$\sim \stackrel{\text{def}}{=} \{(Q, Q \mid \nu a.a^? x.P) \mid P, Q \text{ beliebig}\}.$$

Wir zeigen, daß $\sim$ eine Bisimulation ist. Nimm an, $Q \stackrel{\alpha}{\Longrightarrow} Q'$. Wenn $\alpha = bc$, dann gilt $Q \twoheadrightarrow bc \mid Q'$, also auch

$$Q \mid \nu a.a^? x.P \twoheadrightarrow Q \mid \nu a.a^? x.P \mid bc$$

und $(Q \mid bc, Q \mid bc \mid \nu a.a^? x.P)$ ist in $\sim$. Der Fall $\alpha = b(\nu c)$ ist analog. Falls $\alpha = b^? c$, so gilt $Q \mid bc \twoheadrightarrow Q'$, also auch

$$Q \mid bc \mid \nu a.a^? x.P \twoheadrightarrow Q' \mid \nu a.a^? x.P.$$

Außerdem gilt wieder $(Q', Q' \mid \nu a.a^? x.P) \in \sim$. Dies zeigt, daß $\sim$ eine Simulation ist. Ein im wesentlichen analoges Argument zeigt, daß $\sim^{-1}$ ebenfalls eine Simulation ist, $\sim$ also seine Bisimulation ist. Mit $Q = \mathsf{0}$ folgt, daß $\mathsf{0} \sim \nu a.a^? x.P$ und damit die Behauptung.

Aufgabe 11.3 Zeigen Sie, daß das Inverse $\sim^{-1}$ der Relation $\sim$ aus Beispiel 11.5 eine Simulation ist.

Die Bedeutung von $\approx$ als Prozeß-Äquivalenz wird durch den nächsten Satz deutlich, der aussagt, daß äquivalente Prozesse in jedem Kontext austauschbar sind.

Satz 11.6 $\approx$ ist eine *Kongruenz*. Das heißt:

1. $\approx$ ist eine Äquivalenzrelation.
2. $P \approx Q$ impliziert $C[P] \approx C[Q]$, für jeden Kontext C.
3. $P \approx Q$ impliziert $SP \approx SQ$, für jeden Substitution S von Namen für die freien Namen von P und Q.

Zum Beweis siehe [6].

Der Zusammenhang von Bisimulation und operationeller Äquivalenz wird im nächsten Satz erklärt.

Definition. Eine Relation $\sim$ ist *abgeschlossen bzgl. Reduktion*, wenn $P \sim Q$ und $P \twoheadrightarrow P'$ impliziert, daß ein Q' existiert, so daß $Q \twoheadrightarrow Q'$ und $P' \sim Q'$.

Satz 11.7 $\approx$ ist die größte Kongruenz, so daß gilt

1. $P \approx Q$ und $P \Downarrow_?$ impliziert $Q \Downarrow_?$.
2. $\approx$ ist abgeschlossen bzgl. Reduktion.

Demgegenüber läßt sich operationelle Äquivalenz auch folgendermaßen erklären:

Satz 11.8 $\cong_?$ ist die größte Kongruenz, so daß gilt:

1. $P \approx Q$ und $P \Downarrow_?$ impliziert $Q \Downarrow_?$.

Zum Beweis siehe wieder [6].

Im Vergleich zur operationellen Äquivalenz kommt also bei der Bisimulation noch der Aspekt der Abgeschlossenheit bezüglich Reduktion dazu. Intuitiv entspricht das der Fähigkeit eines Beobachters, einen Prozeß an beliebiger Stelle anzuhalten, und beliebig oft neu zu starten, um Experimente zu wiederholen.

Kapitel 12

Objekt-Orientierte Programmierung

Zur Motivation und allgemeinen Behandlung des OOP siehe die Folienkopien.

12.1 Darstellung in π_a

Betrachte ein Objekt

$$\begin{array}{l}\{i_1 = v_1, \ldots, i_k = v_k \; ; \\ m_1 = P_1, \ldots, m_k = P_l\}\end{array}$$

mit Instanzvariablen $i_1, \ldots, i_k$ und Methoden $m_1, \ldots, m_l$. Da Objekte Identität haben, liegt es nahe, ein Objekt als einen Namen ($self$) zu implementieren, von dem Meldungen, die an das Objekt geschickt werden, gelesen werden. Eine Meldung bestimmt, welche Methode des Objekts aufgerufen wird. Da Methoden $self$ als ersten impliziten Parameter übergeben bekommen, können Sie das umschließende Objekt identifizieren, z. B. um es an eine andere Methode weiterzureichen. Instanzvariablen sind von außen nicht sichtbar, Sie werden deshalb als normale Variablen dargestellt, deren Sichtbarkeitsbereich auf das Objekt selbst beschränkt ist. Zusammengefaßt ergibt sich folgende Kodierung.

$$
\begin{array}{l}
\nu self.\nu i_1.\ \ldots\ \nu i_k. \\
(\quad i_1 v_1 \mid \ldots \mid i_k v_k \mid \\
\quad !self?msg. \\
\quad\ (\quad [msg = m_1] P_1\ self \\
\quad\ \ \mid \quad \vdots \\
\quad\ \ \mid \quad [msg = m_l] P_l\ self \\
\quad\ ) \\
)
\end{array}
$$

12.2 Literatur

Empfehlenswert sind der Übersichtartikel von Wegner [22] und die einführenden Kapitel des Buchs von Goldberg und Robson über Smalltalk [4]

Kapitel 13

Nebenläufige Programmierkonzepte

Siehe dazu die Folienkopien.

Literaturverzeichnis

[1] Samson Abramsky, Pasquale Malacaria, and Radha Jagadessan. Full abstraction for PCF (extended abstract). In *Proc.Theoretical Aspects of Computer Software*, April 1994. Springer LNCS 789.

[2] John W. Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 21:613–641, 1978.

[3] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *Fourth Symposium on Functional Programming Languages and Computer Architecture*, pages 26–38. ACM, September 1989. London.

[4] Adele Goldberg and David Robson. *Smalltalk-80; The Language and Its Implementation.* Addison-Wesley, 1983.

[5] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Series. MIT Press, 1992.

[6] Keiho Honda and Nobuko Yoshida. On reduction-based process semantics. In *Proc. 13th Conf. on Foundations of Softawre Technology and Theoretical Computer Science*, pages 373–387, December 1993.

[7] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[8] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[9] Kenneth E. Iverson. *A Programming Language.* John Wiley and SOns, 1962.

[10] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987. Lecture Notes in Computer Science 247.

[11] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 24–35, June 1994.

[12] Robin Milner. The polyadic π-calculus: A tutorial. Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Edinburgh University, October 1991.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I + II. *Information and Computation*, 100:1–77, 1992.

[14] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[15] Gordon D. Plotkin. Structural operational semantics. Report DAIMI-FN19, Computer Science Department, Aarhus University, 1981.

[16] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

[17] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, June 1992. IEEE Computer Society Press.

[18] Simon Thompson. *Type Theory and Functional Programming*. International Computer Science Series. Addison-Wesley, Reading, Massachusetts, 1991.

[19] A. van Wijngaarden, B. J. Mailloux, J. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. Meertens, and R. G. Fisker. Revised report on the algorithmic language Algol 68. *Acta Informatica*, 5:1–236, 1975.

[20] Torsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th International Symposium on Computer Architecture*, May 1992.

[21] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, April 1990.

[22] Peter Wegner. Dimesions of object-based language design. In *Proc. OOPSLA'87*, pages 168–181, 1987.