Inhalt

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- die grundlegenden Konzepte funktionaler Programmiersprachen kennen. Insbesondere sollten Sie *statisches Binden* erklären und die Bedeutung von *closures* darstellen können.
- die Typen von Ausdrücken ermitteln können,
- den Begriff *generische Instanz* erklären können sowie den Unterschied zwischen generischen und nicht-generischen Typvariablen darstellen und deren Bedeutung erklären können,
- ein polymorphes Typinferenzsystem für eine funktionale Sprache erklären können, d. h. die Typregeln angeben und ihre Funktionsweise erläutern können,
- die Unifikation von (Typ-)Termen beherrschen und den Algorithmus zur automatischen Typinferenz erklären und anwenden können,
- einen Interpreter für eine funktionale Sprache angeben können und dabei die Aufgabe von zyklischen Umgebungen beschreiben können,
- die SECD-Maschine einschließlich ihrer Befehle und Transitionsregeln erläutern können,
- eine funktionale Sprache in SECD-Code übersetzen können.

Kapitel 7

Übersetzung funktionaler Programmiersprachen

In diesem Kurs wurden bisher als Beispiele vornehmlich Elemente aus imperativen Programmiersprachen betrachtet. Ein Programm in einer solchen Sprache besteht im Wesentlichen aus einer Folge von Anweisungen, die zur Laufzeit Zustandstransformationen bewirken. Von Ein-/Ausgaben einmal abgesehen, ist dann das Ergebnis eines solchen Programms üblicherweise ein Teil des resultierenden Zustandes.

Dies ist in funktionalen Programmiersprachen völlig anders: Dort existieren nämlich Anweisungen überhaupt nicht, sondern lediglich Ausdrücke, und ein funktionales Programm ist eigentlich nichts anderes als ein Ausdruck (der mitunter allerdings sehr komplex werden kann). Zur Laufzeit werden entsprechend auch keine Zustände verändert, und das Ergebnis eines funktionalen Programms ist ganz einfach dessen Wert. Man muss in funktionalen Sprachen natürlich auch in irgendeiner Form Kontrollstrukturen realisieren. Dies geschieht einerseits über *Rekursion* und andererseits durch *Funktionen höherer Ordnung*; das sind Funktionen, die als Argumente und/oder als Ergebnis selbst wieder Funktionen haben. Man sagt manchmal auch, Funktionen besitzen „volle Bürgerrechte“, um zu unterstreichen, dass sie genauso wie alle anderen Werte verwendet werden können und nicht einen eingeschränkten Status besitzen wie in vielen imperativen Sprachen. Es ist gerade diese Allgemeinheit von Funktionen, die besondere Techniken bei der Implementierung funktionaler Sprachen erfordert.

Ein weiteres Charakteristikum moderner funktionaler Sprachen ist der hohe Stellenwert der Typisierung. So besitzt einerseits jeder Wert einen eindeutig bestimmten Typ – diese Eigenschaft nennt man *strenge Typisierung* –, andererseits lässt sich der Typ eines jeden Ausdrucks vor dessen Auswertung allein durch Betrachten seiner syntaktischen Struktur bestimmen – diesen Vorgang nennt man *statische Typprüfung*. Ein wesentliches Merkmal von Typsystemen funktionaler Sprachen ist die Existenz *polymorpher Typen*. Das bedeutet, dass eine Funktion nicht nur für einen bestimmten Wertebereich definiert werden kann, sondern auch für eine ganze Klasse von Typen. Damit verringert sich der Programmieraufwand, und es erhöht sich die Wiederverwendbarkeit von Software: Viele Funktionen (z. B. zum Sortieren) werden nur einmal definiert und können dann für Werte beliebigen Typs eingesetzt werden.

Zunächst geben wir in Abschnitt 7.1 eine kurze Einführung in Teile von ML, einer neueren funktionalen Sprache. In Abschnitt 7.2 beschreiben wir ein polymorphes Typsystem, und wir zeigen, wie man automatisch Typen für beliebige Ausdrücke

herleiten kann. Die Typinferenz fällt in die Phase der semantischen Analyse. Im Weiteren können wir dann von korrekt typisierten Programmen ausgehen. In Abschnitt 7.3 stellen wir einen Interpreter für ML vor, und in Abschnitt 7.4 zeigen wir dann die Übersetzung in eine abstrakte Stack-Maschine, die SECD-Maschine.

Die Beschreibung funktionaler Programmierung und der betrachteten Sprache ML, wie auch die Darstellung der Übersetzung, ist durch den Rahmen eines einzigen Kapitels sehr eng begrenzt und muss somit zwangsläufig eine Reihe interessanter Details ausklammern. So verzichten wir hier auf die Erläuterung der für das Programmieren äußerst wichtigen Datentypkonstruktoren und des damit verbundenen pattern matchings. Auf weiterführende Literatur zu funktionalen Programmiersprachen und deren Übersetzung gehen wir am Ende diese Kapitels ein.

7.1 ML

Ausdrücke, Werte und Funktionen

Ein *funktionales Programm* ist, wie schon erwähnt, im Grunde nichts anderes als ein mehr oder weniger komplexer Ausdruck. *Ausdrücke* werden durch Anwendung von Funktionssymbolen auf Ausdrücke und Werte gebildet. Dies können beispielsweise einfache arithmetische Ausdrücke sein wie:

3*(4+1)

oder aber auch Ausdrücke, die kompliziertere Berechnungen und Strukturen beschreiben, z. B.

map size ["Eine", "Liste", "von", "strings"]

Hier ist *map* eine Funktion (höherer Ordnung), die die Funktion *size* zur Berechnung der Länge eines string-Wertes auf alle Elemente einer Liste anwendet.

Ausdrücke bezeichnen Werte, und diese werden durch die Implementierung einer funktionalen Sprache berechnet. *Werte* sind somit Ausdrücke, die nicht weiter ausgewertet werden können. Die Ergebnisse der obigen Ausdrücke sind der integer-Wert 15 bzw. der Listenwert [4, 5, 3, 7].

Funktionen sind nun ebenfalls Werte, genauso wie integers, strings oder Listen. Zunächst gibt es natürlich vordefinierte Funktionen wie z. B. * oder *size*, darüber hinaus kann man aber auch Ausdrücke formulieren, die Funktionen bezeichnen. In ML lautet die Syntax für einen Funktionsausdruck:

fn *param* $\Rightarrow$ *exp*

Diese Form nennt man auch *funktionale Abstraktion*. Dabei steht *param* für den (oder die) Parameter der Funktion, und *exp* ist der definierende Ausdruck. Beispiels-

weise lauten die Definitionen für Nachfolgerfunktion und Maximum auf den ganzen Zahlen:

fn $x \Rightarrow x+1$
fn $(x, y) \Rightarrow$ **if** $x>y$ **then** x **else** y

Die Applikation von Funktionen erfolgt in der Regel durch Präfix-Notation, den Nachfolger von 7 kann man also durch den Ausdruck

(**fn** $x \Rightarrow x+1$) (4+3)

berechnen.[1] Wie die Auswertung dieses Ausdrucks genau erfolgt, werden wir noch sehen. Vereinfacht kann man sich vorstellen, dass der definierende Ausdruck ausgewertet wird, nachdem alle Vorkommen des Parameters durch den Wert des Arguments ersetzt worden sind. Das heißt, Argumente werden ausgewertet, bevor sie an Parameter gebunden werden. Diese Art der Parameterübergabe nennt man *call-by-value*.

Die obige Notation wird sicherlich sehr bald lästig werden. Man möchte natürlich eine häufig benötigte Funktion nur einmal, an einer Stelle, definieren und dann unter Verwendung eines geeigneten Namens an vielen anderen Stellen benutzen. Dies werden wir im Anschluss betrachten. Es bleibt noch zu erwähnen, dass man die gezeigten Funktionsausdrücke mangels Namensgebung auch *anonyme Funktionen* nennt.

Variablen, Bindungen und Umgebungen

Ein jeder Wert kann benannt werden, um über seinen Namen in anderen Ausdrücken verwandt zu werden. Dies wird durch die Form

let val *var* = *exp'* **in** *exp* **end**

ermöglicht. Das Ergebnis ergibt sich durch die Auswertung von *exp*, wobei darin zuvor alle Vorkommen von *var* durch den Wert von *exp'* ersetzt werden. Wir können also beispielsweise Funktionen oder auch Konstanten definieren:

```
let   val suc = fn x => x+1          let   val c = 4
in                                   in
      suc (suc 7)                          c*c+c
end                                  end
```

Zur besseren Lesbarkeit kann man Funktionsdefinitionen **val** *var* = **fn** *arg* $\Rightarrow$ *exp* auch kurz als **fun** *var arg* = *exp* notieren. Das heißt, wir können den linken Ausdruck wie folgt umschreiben:

---

[1] Die Applikation hat die syntaktisch stärkste Bindungskraft und ist linksassoziativ, d. h., ein Ausdruck *f g x+y* wird implizit als ((*f g*) *x*)+*y* geklammert.

```
let   fun suc x = x+1
in
      suc (suc 7)
end
```

Wir wollen nun die Semantik eines **let**-Ausdrucks präzise erklären. Dazu benötigen wir die beiden Konzepte *Bindung* und *Umgebung*. Eine Bindung ist ein Paar (Variable, Wert), und eine Umgebung ist eine Liste von Bindungen. Nun werden Ausdrücke stets in einer Umgebung ausgewertet, wobei Variablen durch die für sie in der Umgebung gespeicherten Werte ersetzt werden. Dabei werden Umgebungslisten von vorne nach hinten durchsucht. Damit wird ein Ausdruck

let val *var* = *exp*' **in** *exp* **end**

in einer Umgebung U nun wie folgt ausgewertet (wir notieren mit $x \cdot L$ das Anfügen eines Elementes x vorne an eine Liste L):

1. Der Ausdruck *exp*' wird in U zu einem Wert v ausgewertet.
2. Der Ausdruck *exp* wird in der Umgebung $(var, v) \cdot U$ ausgewertet.

Betrachten wir als Beispiel die Berechnung des Ausdrucks

```
let   val x = 3
in
      (let val x = x*x in x+x end) – x
end
```

in der leeren Umgebung. Zunächst wird also 3 ausgewertet. Das Ergebnis ist offensichtlich 3. Danach ist dann in der Umgebung $U_1 = [(x, 3)]$ die Differenz (**let val** x = x*x **in** x+x **end**) – x auszuwerten. Dazu ermitteln wir zunächst den linken Teilausdruck. Also berechnen wir x*x in der Umgebung U_1 (was aufgrund der Bindung des Wertes 3 an x zum Ergebnis 9 führt) und müssen dann x+x in der Umgebung $U_2 = (x, 9) \cdot U_1 = [(x, 9), (x, 3)]$ auswerten, das Ergebnis ist also 18. Nun benötigen wir den Wert des rechten Teilausdrucks. Dazu müssen wir x in der Umgebung U_1 suchen (und nicht in U_2, denn U_2 galt ja nur für den Ausdruck x+x). Mit dem Wert 3 ergibt sich also das Endergebnis 15.

Man sieht an diesem Beispiel, wie innere (spätere) Definitionen vorhergehende verdecken. Dies wird dadurch erreicht, dass sowohl das Suchen nach Variablen als auch das Anfügen neuer Definitionen am Anfang der Umgebung erfolgt. Man beachte jedoch, dass Variablendefinitionen niemals im Sinne imperativer Sprachen „überschrieben" werden, so ist z. B. nach der Berechnung des inneren **let**-Ausdrucks die erste Definition für x wieder sichtbar.

Das vorangegangene Beispiel zeigt auch, wie man mehr als nur eine Definition in einem Ausdruck bewirken kann: Da die beschriebene **let**-Form selbst ein Ausdruck ist, kann man durch die nachfolgend gezeigte Schachtelung die Verwendung einer Reihe von Definitionen in einem Ausdruck erreichen (rechts ist eine abkürzende, äquivalente Notation gezeigt). Damit kann man sich ein funktionales Programm vereinfacht als **let**-Ausdruck vorstellen: Mit den einzelnen Variablendefinitionen wer-

den Funktionen und Konstanten definiert, und *exp* übernimmt die Rolle eines „Hauptprogramms“, wie man es aus imperativen Sprachen kennt.

```
let   val var1 = exp1 in
let   val var2 = exp2 in
      ...
let   val varn = expn
in
      exp
end   ... end end
```

```
let   val var1 = exp1
      val var2 = exp2
      ...
      val varn = expn
in
      exp
end
```

Mit der Semantik für **let**-Ausdrücke lässt sich auch die Bedeutung von Funktionsapplikationen genau erklären. Ein Ausdruck

(**fn** *var* ⇒ *exp*) *exp*’

ist nämlich vollkommen äquivalent zum Ausdruck

let val *var* = *exp*’ **in** *exp* **end**

Das bedeutet, dass eine Bindung des Parameters an das Argument in die aktuelle Umgebung eingefügt wird und darin dann der definierende Ausdruck ausgewertet wird. (Die Erweiterung auf mehrere Parameter kann man sich leicht vorstellen. Man beachte jedoch, dass sämtliche Parameter einer Funktion paarweise verschieden sein müssen.)

Wir müssen an dieser Stelle noch auf ein Phänomen eingehen, das den Wert freier[2] Variablen in Funktionen betrifft. Als Beispiel betrachten wir folgenden Ausdruck:

```
let   val x = 1
      fun plusx y = x+y
      val x = 2
in
      plusx 3
end
```

Wie lautet nun das Ergebnis? Entscheidend ist, ob sich das freie *x* in der Funktionsdefinition für *plusx* auf die erste oder die zweite Definition von *x* bezieht. Die erste Definition ist zum Zeitpunkt der Funktionsdefinition gültig, die zweite Definition ist gültig, wenn die Funktion appliziert wird. In der Tat sind beide Möglichkeiten denkbar, bewährt hat sich allerdings die erste Variante, die auch in fast allen funktionalen Sprachen (bis auf einige LISP-Dialekte) Anwendung findet: Der Wert einer freien Variablen in einer Funktion ist der zum Zeitpunkt der Funktionsdefinition

---

[2] Eine Variable ist in einem Ausdruck *frei*, wenn sie nicht durch eine Abstraktion gebunden ist. Zum Beispiel ist *x* im Ausdruck **fn** *x* ⇒ *x*+*y* gebunden, während *y* frei ist. Im Ausdruck (**fn** *x* ⇒ *x*) *x* tritt *x* sowohl gebunden als auch frei auf. Wir verzichten an dieser Stelle auf eine formale Definition, s. auch Abschnitt 7.2.2.

gültige. Dies nennt man auch *statisches Binden* (*static binding*). Das Ergebnis des obigen Ausdrucks ist demnach 4.

Nun müssen wir dieses Verhalten noch mit der obigen Semantikdefinition in Einklang bringen. Danach würde zunächst die Bindung (x, 1) in der Umgebung gespeichert, dann müsste eine Bindung für die Funktionsdefinition erzeugt werden (*plusx*, *f*). (*f* bezeichne hier den Funktionswert, dessen genaue Struktur wir noch bestimmen.) Schließlich wird darüber eine erneute Bindung für x, nämlich (x, 2), gelegt. In dieser Umgebung U = [(x, 2), (*plusx*, *f*), (x, 1)] wird dann der Ausdruck *plusx* 3 ausgewertet. Dazu wird der Wert von *plusx*, d. h. *f*, ermittelt und auf 3 appliziert. Wie genau muss *f* nun beschaffen sein? Einerseits muss darin sicherlich die Definition der Funktion, **fn** $y \Rightarrow x+y$, enthalten sein. Wenn wir diese aber auf 3 applizieren und den Ausdruck in U auswerten, so wird fälschlicherweise der Wert 2 für x verwandt. Daher muss man sich in *f* zusätzlich noch die zum Zeitpunkt der Definition gerade gültigen Werte der freien Variablen merken. Dies kann man ganz einfach dadurch erreichen, dass man die gesamte gerade aktuelle Umgebung zusammen mit der Funktionsdefinition in *f* ablegt. Das heißt, in unserem Beispiel ist *f* = (**fn** $y \Rightarrow x+y$, [(x, 1)]). Die Auswertung der Applikation erfolgt dann in der mit der Funktion gespeicherten Umgebung, d. h., der definierende Ausdruck wird in der um die Bindung für den (oder die) Parameter erweiterten gespeicherten Umgebung ausgewertet. Im Beispiel also wird $x+y$ in der Umgebung [(y, 3), (x, 1)] ausgewertet und liefert so den korrekten Wert 4. Ein Paar bestehend aus Funktionsdefinition und Umgebung nennt man *Funktionsabschluss* (*closure*). Closures erfüllen eine ganz ähnliche Aufgabe wie die Displays aus Abschnitt 6.1.2.

Wir sind bisher noch nicht auf rekursive Funktionsdefinitionen eingegangen. Die genaue Auswertung rekursiv definierter Funktionen und insbesondere die dabei notwendige Behandlung der Umgebung werden wir in Abschnitt 7.3 untersuchen. An dieser Stelle sei nur erwähnt, dass rekursive Funktionsdefinitionen natürlich möglich sind. Die Fakultätsfunktion wird z. B. definiert durch

fun *fak* x = **if** $x<3$ **then** x **else** x**fak* (x–1)

Dies ist wiederum eine abkürzende Notation für die rekursive Definition

val rec *fak* = **fn** $x \Rightarrow$ **if** $x<3$ **then** x **else** x**fak* (x–1)

Man beachte hier das Schlüsselwort **rec**, das die Verwendung des zu definierenden Bezeichners (*fak*) innerhalb seiner eigenen Definition ermöglicht.

Funktionen höherer Ordnung

Die Möglichkeit, dass Funktionen Parameter und auch Ergebnis von Funktionen sein können, ermöglicht es, mit einem relativ kleinen Satz von Funktionen eine Vielzahl verschiedener Programmieraufgaben zu erledigen. Wenn man beispielsweise immer wieder Elemente aus Listen selektieren muss, so genügt dazu eine einzige Funktion, die u. a. als Parameter das Selektionskriterium, eine boolesche Funktion, hat. Weitere Beispiele sind die eingangs gezeigte Funktion *map* und das ebenfalls

erwähnte generische Sortieren. Viele Funktionen höherer Ordnung (wie auch die soeben genannten Beispiele) operieren auf Datentypen wie Listen, Bäumen usw. Da wir Datentypen hier nicht besprochen haben, müssen wir uns auf die Darstellung einiger weniger elementarer Funktionen höherer Ordnung beschränken.

Die Funktion *twice* nimmt als Argument eine unäre Funktion und appliziert diese zweimal auf einen Wert:

fun *twice* (*f*, *x*) = *f* (*f x*)

Der Ausdruck *twice* (*suc*, 3) ergibt somit 5. In der dargestellten Form nimmt *twice* eine Funktion und einen Wert als Argument, das Ergebnis ist ein Wert. Insbesondere zur einfacheren Definition neuer Funktionen ist es hilfreich, wenn Funktionen wie *twice* selbst auch Funktionen als Ergebnis haben. Mit der alternativen Definition

fun *twice f* = **fn** *x* ⇒ *f* (*f x*)

ist *twice* nun eine Funktion, die lediglich eine Funktion als Parameter hat und eine Funktion als Ergebnis liefert, d. h., eine Applikation von *twice* erfolgt auf eine Funktion (und keinen zusätzlichen Wert), z. B. *twice suc*, und das Ergebnis einer solchen Applikation ist selbst wieder eine Funktion, die dann auf einen Wert angewandt werden kann, also z. B. *twice suc* 3.[3]

Damit kann man dann sehr einfach neue Funktionen definieren:

val *plus2* = *twice suc*
val *fourtimes* = *twice twice*

Wir hatten die Form **fun** *f x* = *e* als Abkürzung für **val** *f* = **fn** *x* ⇒ *e* eingeführt. Diese Notation gilt auch für geschachtelte Abstraktionen, so dass man die zweite Definition von *twice* auch wie folgt notieren kann:

fun *twice f x* = *f* (*f x*)

Diese Form der Definition bezeichnet man nach dem Logiker Haskell B. Curry auch als *currying*.[4] Funktionsdefinitionen mittels currying sind immer dann besonders nützlich, wenn man – wie im Falle von *twice* – durch teilweise Fixierung der Argumente neue Funktionen definieren möchte. Dies betrifft nicht nur Funktionsargumente, betrachten wir z. B. die Definition der Funktion *divides*, die überprüft ob *j* durch *i* teilbar ist:

fun *divides i j* = (*j mod i* = 0)

Mit *divides* kann man sehr leicht die Funktion *even* definieren:

val *even* = *divides* 2

---

[3] Man beachte die Linksassoziativität der Applikation. Das heißt, der Ausdruck wird implizit als (*twice suc*) 3 geklammert.

[4] Eigentlich hat der deutsche Mathematiker Schönfinkel die Notation zuerst erfunden, aber der Begriff „schönfinkeln“ hat sich leider nicht durchsetzen können.

Eine in vielen Sprachen vordefinierte Funktion höherer Ordnung ist die Funktionskomposition. Zu zwei Funktionsargumenten *f* und *g* ergibt beispielsweise in ML der Ausdruck $f \circ g$ die Funktion, die zu einem Wert *x* den Wert $f\ (g\ x)$ berechnet. Auch damit lassen sich weitere Funktionen definieren:

val *odd* = *not* ∘ *even*

Mini-ML

In den folgenden Abschnitten betrachten wir die Sprache *Mini-ML*, deren Syntax durch die Grammatik in Abb. 7.1 definiert ist.

exp →	*con*	*Konstante*
	| *var*	*Variable*
	| **if** *exp* **then** *exp* **else** *exp*	*Fallunterscheidung*
	| *con exp*	*Applikation*
	| *exp con exp*	
	| *exp exp*	
	| **fn** *var* ⇒ *exp*	*Abstraktion*
	| **let val** *var* = *exp* **in** *exp* **end**	**let**-*Ausdruck*
	| **let val rec** *defs* **in** *exp* **end**	*rekursives* **let**
defs →	*var* = *exp*	
	| *var* = *exp* **and** *defs*	

Abb. 7.1. Syntax von Mini-ML

Wir unterscheiden drei Arten von Applikationen: Unäre und binäre vordefinierte Funktionen sowie Applikationen von Abstraktionen. Man sieht dabei, dass Konstanten nicht nur Werte wie integers oder strings umfassen, sondern auch Bezeichner für (vordefinierte) Funktionen (wie z. B. + oder *not*). Zu beachten ist auch, dass Abstraktionen nur über Variablen (und nicht über Tupeln) erlaubt sind. Funktionen mit mehreren Argumenten sind also „geschönfinkelt“ zu definieren. Im rekursiven **let**-Ausdruck werden im Allgemeinen mehrere Gleichungen benötigt (um z. B. wechselseitig rekursive Definitionen zu ermöglichen). Für die definierenden Ausdrücke innerhalb einer *defs*-Folge sind lediglich Abstraktionen erlaubt, d. h., es können nur Funktionen definiert werden und keine anderen Werte.

7.2 Polymorphe Typsysteme und Typinferenz

Jeder Ausdruck in ML hat einen eindeutig bestimmten Typ. Aussagen über den Typ eines Ausdrucks notieren wir in der Form

$exp : \tau,$

wobei τ einen Typ bezeichnet. Die Tatsache, dass die Zahl 3 vom Typ integer ist, schreiben wir also als $3 : int$. Neben einfachen Typen wie *int* oder *string* kann man mittels *Typkonstruktoren* Typausdrücke bilden, die komplexe Typen bezeichnen. In imperativen Sprachen wie PASCAL stehen Typkonstruktoren z. B. für Records oder Arrays zur Verfügung. Wenn man Typen als Mengen der enthaltenen Werte interpretiert, so realisiert beispielsweise der Record-Konstruktor das kartesische Produkt (wobei Komponenten benannt sind). In funktionalen Sprachen gibt es neben Records auch das einfache (unbenannte) kartesische Produkt, so beschreibt beispielsweise der Ausdruck $int \times bool$ den Typ für (integer, boolean)-Paare.

Da nun Funktionen selbst Werte sind, haben auch diese jeweils einen Typ. Dabei werden Funktionstypen mit Hilfe des „$\rightarrow$"-Typkonstruktors gebildet:

$suc : int \rightarrow int$
$size : string \rightarrow int$

Polymorphismus

In funktionalen Sprachen gibt es über Typen und Typkonstruktoren hinaus noch *Typvariablen* (α, β, …), mit deren Hilfe man *polymorphe Typen* konstruieren kann. Betrachten wir einmal die Identitätsfunktion, die ihr Argument unverändert als Ergebnis liefert:

fun $id\ x = x$

Man kann nun *id* nicht nur auf Zahlen, sondern auch auf Werte anderer Typen anwenden, z. B. auf Wahrheitswerte, aber auch auf komplexe Werte wie Listen oder gar Funktionen. Den Typ von *id* notiert man als

$id : \alpha \rightarrow \alpha$

Dabei steht die Typvariable α für einen beliebigen (jedoch innerhalb des Typausdrucks, festen) Typ. Das heißt, der Typ von *id* umfasst Typen wie $int \rightarrow int$, $bool \rightarrow bool$ oder aber auch $(int \rightarrow int) \rightarrow (int \rightarrow int)$[5]. Letzterer besagt, dass man *id* beispielsweise auch auf die Funktion *suc* anwenden kann.

Hier noch einige weitere Beispiele für polymorphe Funktionsdefinitionen und deren Typen ($\times$ bindet stärker als $\rightarrow$):

---

[5] Da $\rightarrow$ rechtsassoziativ ist, wird dieser Typ üblicherweise auch als $(int \rightarrow int) \rightarrow int \rightarrow int$ notiert.

fun *swap* $(x, y) = (y, x)$	$\alpha \times \beta \to \beta \times \alpha$
fun *ignore* x = "Argument ist futsch"	$\alpha \to string$
fun *twice* $f\, x = f\,(f\, x)$	$(\alpha \to \alpha) \to \alpha \to \alpha$
fun *compose* (f, g) = **fn** $x \Rightarrow f\,(g\, x)$	$(\alpha \to \beta) \times (\gamma \to \alpha) \to \gamma \to \beta$

Nun ist ein *Typsystem* ein logisches System, das Aussagen der Form „der Ausdruck *exp* hat den Typ τ“ (notiert als $exp : \tau$) formalisiert. Derartige Aussagen können über die Axiome und Regeln des logischen Systems bewiesen werden. Ein Typsystem werden wir für eine Sprache wie ML in Abschnitt 7.2.2 entwickeln. In 7.2.3 betrachten wir dann einen Algorithmus, der mit Hilfe der Regeln eines solchen Typsystems zu einem Ausdruck einen Typ ermittelt. Diesen Vorgang nennt man *Typinferenz*. Der vorgestellte Algorithmus hat zudem die Eigenschaft, dass er jeweils den allgemeinsten Typ eines Ausdrucks findet. Zunächst betrachten wir aber in Abschnitt 7.2.1 einige Besonderheiten, die bei der Typbestimmung zu beachten sind.

7.2.1 Bestimmung von Typen

Wir wollen den Typ für das folgende Programmstück ermitteln:

```
let   fun f x = x
in
      f 3
end
```

Es ist unschwer zu erkennen, dass der gesamte Ausdruck den Typ *int* hat, aber wie kann man dies formal herleiten? Man kann sich Typinferenz vereinfacht als das Lösen einer Menge von Gleichungen vorstellen: Zunächst ermittelt man für alle Elemente in einem Ausdruck (d. h. Konstanten, Funktionen und Variablen) den allgemeinsten möglichen Typ. Für Konstanten ist dies ihr definierter Typ, für eine Variable eine Typvariable und für Funktionen ein Typ $\alpha \to \beta$. Diese Informationen notiert man in Form einer Menge von Gleichungen. Für das obige Beispiel erhalten wir also die Gleichungen $type(3) = int$, $type(x) = \alpha$ und $type(f) = \beta \to \gamma$. Dabei muss man zunächst für verschiedene Variablen und Funktionen verschiedene Typvariablen nehmen, da die Verwendung von gleichen Typvariablen ja die Gleichheit der entsprechenden Typen ausdrückt. Darüber kann man aber zu diesem Zeitpunkt noch gar keine Aussagen machen.

Nun fügt man Gleichungen hinzu, die sich aus der Struktur des Ausdrucks ergeben. Diese Gleichungen ergeben sich aus allgemeinen Regeln wie z. B.: „Der Typ des Parameters einer Funktionsdefinition muss gleich dem Argumenttyp der Funktion sein“ oder „In einer Applikation muss der Typ des Arguments mit dem Argumenttyp der Funktion übereinstimmen“. Im obigen Beispiel ergeben sich daraus die zusätzlichen Gleichungen $type(x) = \beta$ bzw. $\beta = int$. Eine ähnliche Regel besagt, dass der Typ des definierenden Ausdrucks einer Funktion gleich dem Ergebnistyp der Funktion sein muss. Daraus folgt für unser Beispiel dann noch die Gleichung $type(x) = \gamma$. Damit kann man direkt folgern, dass $\alpha = \beta = \gamma = int$ gelten muss. Da schließlich noch

der Typ einer Applikation gleich dem Ergebnistyp der applizierten Funktion ist, ergibt sich als Typ des Gesamtausdrucks der Typ *int*.

Ein anderes Beispiel: Die Funktion

fn $f \Rightarrow (f\,3, f\,true)$

ist nicht typisierbar. Dies sollte intuitiv klar sein, denn einerseits verlangt der Ausdruck $f\,3$, dass f den Typ $int \to \alpha$ hat, während andererseits die zweite Applikation für f den Typ $bool \to \alpha$ fordert. Diese beiden Typen sind jedoch nicht miteinander vereinbar, d. h., es gibt keinen Typ für f, der den beiden Anforderungen genügt. Damit aber ist auch die gesamte Abstraktion nicht typisierbar. Insbesondere ist nun auch jede Applikation der Funktion, wie z. B. (**fn** $f \Rightarrow (f\,3, f\,true)$) (**fn** $x \Rightarrow x$) nicht typisierbar. Ein ähnliches Beispiel ist der Ausdruck

```
let   val f = fn x ⇒ x
in
      (f 3, f true)
end
```

In diesem Fall kann man für f sehr wohl einen Typ finden, und zwar $\alpha \to \alpha$, den man in beiden Applikationen zu einem jeweils korrekten Typ instanzieren kann, nämlich $int \to int$ für den Ausdruck $f\,3$ und $bool \to bool$ für den Ausdruck $f\,true$.

Dies ist etwas überraschend, hatten wir doch ausdrücklich erwähnt, dass eine Applikation (**fn** $x \Rightarrow e$) e' dem **let**-Ausdruck **let val** $x = e'$ **in** e **end** entspricht. Warum also kann der **let**-Ausdruck typisiert werden, während man für die entsprechende Applikation keinen Typ finden kann? Dies liegt daran, dass innerhalb des **let**-Ausdrucks die Struktur der Funktion f bereits bekannt ist, während die Typisierung der Abstraktion unabhängig von der bei der Applikation für f einzusetzenden Funktion erfolgen muss. Das bedeutet also, dass die beschriebene Korrespondenz zwischen Applikation und **let**-Ausdruck lediglich für die Auswertung von Ausdrükken gilt. Bei der Typisierung unterscheiden wir dagegen lokal gebundene Variablen von Variablen, die Funktionsparameter sind. Falls also x eine Variable einer Abstraktion ist (wie z. B. in **fn** $x \Rightarrow e$), so müssen alle zugehörigen Vorkommen in e den gleichen Typ haben.[6] Die in dem für x ermittelten Typ enthaltenen Typvariablen können nicht zu verschiedenen Typen instanziert werden; deshalb nennt man sie *nicht-generisch*. Ist dagegen x durch einen **let**-Ausdruck gebunden, so können die Typvariablen aus dem Typ für x durchaus an verschiedenen Stellen zu verschiedenen Typen instanziert werden. Daher nennt man solche Typvariablen *generisch*.

Generische Typvariablen werden auf der äußersten Ebene eines Typausdrucks durch einen Allquantor gebunden, um sie in Typausdrücken von nicht-generischen Typvariablen unterscheiden zu können. So notieren wir beispielsweise den Typ von f aus dem **let**-Ausdruck als $\forall\alpha.\alpha \to \alpha$. Einen solchen Ausdruck nennt man *Typschema*.

---

[6] Das heißt, alle Vorkommen, die durch diese Abstraktion gebunden sind und nicht durch eine andere Abstraktion oder einen **let**-Ausdruck innerhalb von e.

Die Instanzierung eines Typschemas erfolgt durch das Ersetzen quantifizierter Variablen durch Typen. So erhält man z. B. den Typ $int \to int$, indem man die Variable α durch den Typ *int* ersetzt. Einen Typausdruck, der durch das Ersetzen von gebundenen Variablen entstanden ist, nennt man *generische Instanz* (dementsprechend bezeichnet man das Ergebnis einer Instanzierung über freie Typvariablen auch als *nicht-generische Instanz*).

7.2.2 Ein polymorphes Typinferenzsystem

Ein *Typsystem* ordnet einem Ausdruck in einer Programmiersprache *S* einen Typ zu, d. h. einen Ausdruck in einer Sprache von Typen τ. Um ein Typsystem genau beschreiben zu können, definieren wir zunächst die Sprachen *S* und τ. Für *S* wählen wir Mini-ML.

Mit einer gegebenen Menge von Typvariablen *TV*, mit $\alpha \in TV$, ist die Sprache der Typen (τ) und die der Typschemata (σ) über folgende Grammatik definiert. Der Einfachheit halber ignorieren wir zunächst Tupel. Entsprechende Erweiterungen sprechen wir unten noch kurz an.

τ	$\to int \mid bool \mid string$	*einfache Typen*
	$\mid \alpha$	*Typvariable*
	$\mid \tau \to \tau$	*Funktionstyp*
σ	$\to \tau$	*Typ*
	$\mid \forall\alpha.\sigma$	*Typbindung*

Mit dieser Definition können Quantoren nicht innerhalb von Typausdrücken verwandt werden (dies nennt man auch flachen Polymorphismus (*shallow polymorphism*)). Daher können wir jedes Typschema auch mit einem Quantor gefolgt von allen gebundenen Variablen notieren: $\forall\alpha_1 \ldots \alpha_n.\tau$. Man beachte, dass ein Typ immer gleichzeitig auch ein Typschema ist.

Bevor wir die Regeln des Typsystems vorstellen, müssen wir den Begriff der generischen Instanz noch präzise definieren. Vereinfacht gesagt, erhält man aus einem Typschema eine Instanz, indem man gebundene Typvariablen durch Typausdrücke ersetzt. Beispielsweise ist der Typ $int \to int$ eine Instanz des Typschemas $\forall\alpha.\alpha \to \alpha$, die man durch Ersetzung der Typvariablen α durch den Typ *int* erhält. Zunächst benötigen wir eine Substitutionsoperation auf Typen und den Begriff der freien/gebundenen Variablen eines Typausdrucks.

Definition 7.1: Die Menge der *freien* und *gebundenen Typvariablen* eines Typausdrucks sind:

$$
\begin{aligned}
FV(\tau) &= \varnothing \quad \text{für } \tau \in \{int, bool, string\} \\
FV(\tau \to \tau') &= FV(\tau) \cup FV(\tau') \\
FV(\forall\alpha_1 \ldots \alpha_n.\tau) &= FV(\tau) - \{\alpha_1, \ldots, \alpha_n\} \\
FV(\alpha) &= \{\alpha\}
\end{aligned}
$$

$GV(\forall\alpha_1\ldots\alpha_n.\tau) = \{\alpha_1, \ldots, \alpha_n\} \cap FV(\tau)$
$GV(\tau) = \varnothing$ □

Beispiel: $FV(\forall\alpha.\beta \to \alpha \to int) = \{\beta\}$ und $GV(\forall\alpha.\beta \to \alpha \to int) = \{\alpha\}$.

Mit $[\tau/\alpha]\tau'$ wird die *Substitution* eines Typs τ für eine Typvariable α in einem Typ τ' notiert. Die Substitution ist induktiv über die Struktur von Typtermen definiert. (Die folgende Definition schließt die Substitution für Typschemata σ aus; dieser Fall wird in der anschließenden Definition der generischen Instanz nicht benötigt.)

Definition 7.2: Die Substitution eines Typs τ für eine Typvariable α in einem Typ τ' ist wie folgt definiert:

$[\tau/\alpha]\tau' = \tau'$ falls $\tau' \in \{int, bool, string\}$
$[\tau/\alpha]\alpha = \tau$
$[\tau/\alpha]\beta = \beta$ falls $\alpha \neq \beta$
$[\tau/\alpha](\tau_1 \to \tau_2) = ([\tau/\alpha]\tau_1) \to ([\tau/\alpha]\tau_2)$ □

Eine Folge von Substitutionen $[\tau_1/\alpha_1](\ldots([\tau_n/\alpha_n]\tau)\ldots)$ notieren wir auch als $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau$. Beispiel: $[int \to \alpha/\alpha, \gamma/\beta](\beta \to \alpha) = \gamma \to (int \to \alpha)$. Damit können wir nun definieren:

Definition 7.3: Ein Typschema $\sigma' = \forall\beta_1\ldots\beta_m.\tau'$ ist *generische Instanz* des Typschemas $\sigma = \forall\alpha_1\ldots\alpha_n.\tau$ (notiert als $\sigma' \prec \sigma$), falls $\tau' = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau$ für beliebige Typen $\tau_1, \ldots, \tau_n$ und falls für alle β_i gilt: $\beta_i \notin FV(\sigma)$. □

Es ist also z. B.

$$\forall\gamma.\beta \to (\beta \to \gamma) \prec \forall\alpha.\beta \to \alpha$$

denn $\tau' = \beta \to (\beta \to \gamma) = [\beta \to \gamma/\alpha](\beta \to \alpha)$ und $\gamma \notin FV(\forall\alpha.\beta \to \alpha) = \{\beta\}$.

Andererseits gilt *nicht*:

$$\forall\beta\gamma.\beta \to (\beta \to \gamma) \prec \forall\alpha.\beta \to \alpha$$

da $\beta \in FV(\forall\alpha.\beta \to \alpha)$.

Die Regeln des Typinferenzsystems betreffen Aussagen der Form $\Gamma \rhd e : \tau$, die ausdrücken, dass unter den Typannahmen in Γ der Ausdruck e den Typ τ hat. Eine *Typannahme* ist ein Paar, bestehend aus Variable (oder Konstante) und Typschema, und eine Menge von Typannahmen ist eine partielle Funktion, die Variablen auf Typschemata abbildet. Beispielsweise ist $\Gamma_1 = \{3 \mapsto int,\ y \mapsto bool\}$ eine Typannahme, die für 3 den Typ *int* und für *y* den Typ *bool* angibt. Typannahmen können erweitert oder auch abgeändert werden: So bezeichnet $\Gamma\{x \mapsto \sigma\}$ die Funktion, die für *x* das Typschema σ liefert und ansonsten genauso wie Γ definiert ist. Im Folgenden werden wir des Öfteren auch die Schreibweise $\Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\}$ als Abkürzung für $(\ldots(\Gamma\{x_1 \mapsto \sigma_1\})\ldots)\{x_n \mapsto \sigma_n\}$ verwenden. Es ist also beispielsweise $\Gamma_1\{z \mapsto string, y \mapsto int\} = \{3 \mapsto int, z \mapsto string, y \mapsto int\}$.

Nun bezeichnet *dom*(Γ) die Menge der Variablen, für die Γ definiert ist. Damit werden die freien Variablen einer Typannahme definiert als:

$$FV(\Gamma) = \bigcup_{x \in \mathrm{dom}(\Gamma)} FV(\Gamma(x))$$

Die Regeln des Typsystems haben nun die Form:

$$\frac{A_1 \quad A_2 \quad \ldots \quad A_n}{B} \quad C$$

Die A_i heißen *Voraussetzungen* der Regel, und *B* ist die *Folgerung* der Regel; *C* ist eine logische Formel. Eine solche Regel besagt anschaulich: Wenn alle Voraussetzungen hergeleitet werden können und zusätzlich noch die Bedingung *C* erfüllt ist, so ist auch die Folgerung herzuleiten. Eine Regel, in der n=0 ist, nennt man auch ein *Axiom*. Das Typsystem für Mini-ML ist in Abb. 7.2 dargestellt.

Die Inferenzregeln sind wie folgt zu lesen: Zunächst besagen die Axiome VAR und CON, dass sich der Typ einer Variablen bzw. einer Konstanten aus der Typannahme ergeben muss. Annahmen für Variablen werden z. B. in den Regeln LET und ABS erzeugt, Annahmen für Konstanten müssen der Inferenz von Anfang an „mitgegeben“ werden.

Die COND-Regel setzt voraus, dass die Bedingung *e* den Typ *bool* und die beiden Alternativen e_1 und e_2 den gleichen Typ τ haben. Dann kann man folgern, dass der Ausdruck **if** *e* **then** e_1 **else** e_2 den Typ τ hat.

Regel APP beschreibt, dass man einen Ausdruck *e* nur dann auf einen anderen Ausdruck *e*’ applizieren darf, wenn *e* eine Funktion ist, d. h. einen Typ der Form $\tau' \to \tau$ hat, und wenn der Typ des Arguments gleich τ’ ist. Dann hat der Ausdruck *e e’* den Typ τ.

In Regel ABS sieht man, dass unter den Annahmen in Γ für eine Abstraktion der Typ $\tau' \to \tau$ hergeleitet werden kann, wenn unter der erweiterten Typannahme $\Gamma\{x \mapsto \tau'\}$, d. h. unter den Annahmen in Γ und der Annahme, dass *x* den Typ τ’ hat, für den Rumpf *e* der Abstraktion der Typ τ hergeleitet werden kann. Zu beachten ist hier, dass τ und τ’ Typen sein müssen und keine Typschemata.

Die Regel LET besagt: Wenn für *e*’ das Typschema σ herleitbar ist und wenn man für *e* unter der Annahme, dass *x* vom Typschema σ ist, den Typ τ zeigen kann, so hat der Ausdruck **let val** *x* = *e*’ **in** *e* **end** den Typ τ.

Die LETREC-Regel verlangt nach einigen Erklärungen. Zunächst sollte klar sein, dass in der Typannahme Γ’’ für die Herleitung des Typs von *e* die Annahmen für sämtliche n Variablen enthalten sein müssen. Da die Ausdrücke e_i wechselseitig rekursiv sein können, erwartet man für deren Herleitung jeweils eine gleichermaßen erweiterte Typannahme. Anstelle von Typschemata muss aber ein geeignet instanzierter Typ angenommen werden, da die Typvariablen der Typen für die x_i nicht generisch in den Ausdrücken e_i sind (sondern lediglich in *e*). Dieses Erfordernis ergibt sich z. B. daraus, dass die rekursive Verwendung einer Variablen in einer Funktionsdefini-

CON $$\frac{}{\Gamma \rhd c : \Gamma(c)}$$

VAR $$\frac{}{\Gamma \rhd v : \Gamma(v)}$$

COND $$\frac{\Gamma \rhd e : bool \quad \Gamma \rhd e_1 : \tau \quad \Gamma \rhd e_2 : \tau}{\Gamma \rhd \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 : \tau}$$

APP $$\frac{\Gamma \rhd e : \tau' \to \tau \quad \Gamma \rhd e' : \tau'}{\Gamma \rhd e\, e' : \tau}$$

ABS $$\frac{\Gamma\{x \mapsto \tau'\} \rhd e : \tau}{\Gamma \rhd \textbf{fn } x \Rightarrow e : \tau' \to \tau}$$

LET $$\frac{\Gamma \rhd e' : \sigma \quad \Gamma\{x \mapsto \sigma\} \rhd e : \tau}{\Gamma \rhd \textbf{let val } x = e' \textbf{ in } e \textbf{ end} : \tau}$$

LETREC $$\frac{\Gamma' \rhd e_i : \tau_i \quad \Gamma'' \rhd e : \tau}{\Gamma \rhd \textbf{let val rec } defs \textbf{ in } e \textbf{ end} : \tau}$$

wobei $defs = x_1 = e_1$ **and** $\ldots$ **and** $x_n = e_n$

$$\Gamma' = \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}$$

$$\Gamma'' = \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\}$$

und $\tau_i \prec \sigma_i$ mit $GV(\sigma_i) \cap FV(\Gamma) = \varnothing$

GEN $$\frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall \alpha.\sigma} \quad \alpha \notin FV(\Gamma)$$

SPEC $$\frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \sigma'} \quad \sigma' \prec \sigma$$

Abb. 7.2. Polymorphes Typinferenzsystem für Mini-ML

tion den gleichen Typ besitzen muss wie die definierte Variable.[7] Die zusätzliche Bedingung, dass in Typschemata gebundene Variablen nicht frei in Γ vorkommen dürfen, ist prinzipiell Voraussetzung für die Generalisierung von Typvariablen, s. dazu auch die folgende Regel GEN.

Mit der Regel GEN kann eine Typvariable generalisiert werden, sofern sie nicht frei in der Typannahme vorkommt. Diese Bedingung kann man sich wie folgt klarmachen: Bei der Typprüfung einer Abstraktion **fn** $x \Rightarrow e$ ist die für x verwandte Typvariable α im Ausdruck e nicht generisch. Nun erscheint α während der Typprüfung von e in der dazugehörigen Typannahme (dies erfordert die Regel ABS). Also dürfen freie Variablen in Typannahmen nicht generalisiert werden.

---

[7] Diese Einschränkung resultiert aus der Tatsache, dass allgemeine Typinferenz für polymorphe Rekursion unentscheidbar ist.

Schließlich beschreibt die Regel SPEC die Instanzierung eines Typschemas: Ist für *e* das Schema σ herleitbar, so ist es auch jedes Typschema σ', das generische Instanz von σ ist.

Betrachten wir nun einige Beispiele. Wir wollen zunächst zeigen, dass die Funktion **fn** $x \Rightarrow x$ den Typ $\forall\alpha.\alpha \to \alpha$ hat, d. h., wir müssen innerhalb des Typsystems beweisen, dass $\{\} \rhd$ **fn** $x \Rightarrow x : \forall\alpha.\alpha \to \alpha$ gilt. Mit $\Gamma = \{x \mapsto \alpha\}$ folgt zunächst aus Regel VAR

$$\Gamma \rhd x : \alpha$$

Die Typannahme in Γ wird in der Regel ABS verwandt (sie wird sozusagen „aufgebraucht"), und es ergibt sich

$$\{\} \rhd \textbf{fn}\ x \Rightarrow x : \alpha \to \alpha$$

Jetzt kann man α generalisieren, da α nicht in der Typannahme vorkommt, d. h., mit der Regel GEN erhält man die gewünschte Aussage.

Nun wollen wir die Applikation dieser Funktion auf einen integer-Wert typisieren. Dazu kann man den generischen Typ der Funktion spezialisieren. Gemäß Definition 7.3 gilt z. B.:

$$int \to int \prec \forall\alpha.\alpha \to \alpha$$

Daher kann man mit der Regel SPEC folgern:

$$\{\} \rhd \textbf{fn}\ x \Rightarrow x : int \to int$$

Die vorangegangenen Typherleitungen können genauso unter erweiterten Typannahmen vorgenommen werden, sofern diese für *x* keinen anderen Typ angeben und α nicht enthalten. Also können wir insbesondere auch

$$\{3 \mapsto int\} \rhd \textbf{fn}\ x \Rightarrow x : int \to int$$

zeigen. Unter der gleichen Annahme erhalten wir mit der Regel CON

$$\{3 \mapsto int\} \rhd 3 : int$$

Diese beiden Aussagen kann man nun als Voraussetzungen für die Regel APP verwenden und damit dann

$$\{3 \mapsto int\} \rhd (\textbf{fn}\ x \Rightarrow x)\ 3 : int$$

herleiten.

Abschließend wollen wir noch die Typisierung des **let**-Ausdrucks

```
let   val f = fn x ⇒ x
in
      (f 3, f true)
end
```

innerhalb des Typsystems nachvollziehen. Dazu benötigen wir eine Erweiterung von Mini-ML und dessen Typsystem um Tupel. Für unser konkretes Beispiel reichen Paare aus. Das heißt, dass (*exp*, *exp*) ebenfalls ein Ausdruck von Mini-ML ist und dass $\tau \times \tau$ ein Element der Sprache τ ist. Für das Typsystem führen wir die folgende Regel ein:

$$\text{TUP} \quad \frac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma \rhd e_2 : \tau_2}{\Gamma \rhd (e_1, e_2) : \tau_1 \times \tau_2}$$

Es sei nun $\Gamma = \{3 \mapsto \mathit{int},\ \mathit{true} \mapsto \mathit{bool}\}$ und $\Gamma' = \Gamma\{f \mapsto \forall\alpha.\alpha \to \alpha\}$. Zunächst zeigen wir

$$\Gamma \rhd \textbf{fn}\ x \Rightarrow x : \forall\alpha.\alpha \to \alpha \tag{1}$$

Dies geschieht wie oben, nur in einer erweiterten Typannahme (die α nicht enthält). Mit der Regel VAR kann man den Typ für *f* aus Γ' entnehmen, d. h. es gilt:

$$\Gamma' \rhd f : \forall\alpha.\alpha \to \alpha$$

Daraus erhält man durch die Regel SPEC die beiden Aussagen:

$$\Gamma' \rhd f : \mathit{int} \to \mathit{int}$$
$$\Gamma' \rhd f : \mathit{bool} \to \mathit{bool}$$

Für diese beiden Instanzen kann man nun mit den Regeln CON und APP getrennt die Aussagen

$$\Gamma' \rhd f\,3 : \mathit{int}$$
$$\Gamma' \rhd f\,\mathit{true} : \mathit{bool}$$

nachweisen. Sie dienen als Voraussetzungen für die Regel TUP, und man erhält

$$\Gamma' \rhd (f\,3, f\,\mathit{true}) : \mathit{int} \times \mathit{bool} \tag{2}$$

Schließlich kann man mit (1) und (2) als Voraussetzungen die Regel LET anwenden und zeigt damit:

$$\Gamma \rhd \textbf{let val}\ f = \textbf{fn}\ x \Rightarrow x\ \textbf{in}\ (f\,3, f\,\mathit{true})\ \textbf{end} : \mathit{int} \times \mathit{bool}$$

Selbsttestaufgabe 7.1: Bestimmen Sie die Typen der folgenden Programmstücke. Beschreiben Sie dabei, welche Regeln des Typsystems anzuwenden sind.

(a) (**fn** $x \Rightarrow x$) (**fn** $x \Rightarrow 1$)

(b) (**fn** $x \Rightarrow$ **fn** $y \Rightarrow x$) 2 □

7.2.3 Automatische Typinferenz

Mit dem Inferenzsystem des vorigen Abschnitts können wir nun Typen sozusagen „von Hand“ herleiten. Offen bleibt aber die Frage nach einem Verfahren, das diese Arbeit automatisch erledigt. Einen solchen Algorithmus kann man sich durchaus als eine Methode zum Suchen von Beweisen in dem vorgestellten logischen System

vorstellen. Insbesondere muss der Algorithmus Entscheidungen treffen, welche Regeln wann anzuwenden sind. Das Inferenzsystem erschwert dies insofern, als in machen Situationen mehr als nur eine Regel anwendbar ist. So kann man beispielsweise die GEN-Regel fast immer anwenden. Bei genauem Betrachten des Typsystems fällt jedoch auf, dass die GEN-Regel eigentlich nur zum Generalisieren von Typvariablen in **let**-Ausdrücken benötigt wird und dass man die SPEC-Regel immer nur direkt nach einer CON-Regel oder nach einer VAR-Regel anzuwenden braucht. Daher kann man die beiden Regeln GEN und SPEC auch wegfallen lassen und deren Aufgaben in die übrigen Regeln integrieren. Dies geschieht wie folgt:

1. Generische Instanzen werden direkt in den neuen Regeln VAR' und CON' durch eine entsprechende Bedingung gebildet.
2. Typschemata werden durch eine Funktion *gen* generiert, die relativ zu einer Typannahme Γ für einen Typ τ das allgemeinste Typschema konstruiert.

Die Funktion *gen* ist wie folgt definiert:

$$gen(\Gamma, \tau) = \forall\alpha_1 \ldots \alpha_n.\tau$$
$$\text{wobei } \{\alpha_1, \ldots, \alpha_n\} = FV(\tau) - FV(\Gamma)$$

Man erhält dann das modifizierte Typinferenzsystem, das in Abb. 7.3 dargestellt ist.

CON' $$\frac{}{\Gamma \rhd c : \tau} \quad \tau \prec \Gamma(c)$$

VAR' $$\frac{}{\Gamma \rhd v : \tau} \quad \tau \prec \Gamma(v)$$

COND $$\frac{\Gamma \rhd e : bool \quad \Gamma \rhd e_1 : \tau \quad \Gamma \rhd e_2 : \tau}{\Gamma \rhd \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 : \tau}$$

APP $$\frac{\Gamma \rhd e : \tau' \to \tau \quad \Gamma \rhd e' : \tau'}{\Gamma \rhd e\, e' : \tau}$$

ABS $$\frac{\Gamma\{x \mapsto \tau'\} \rhd e : \tau}{\Gamma \rhd \textbf{fn } x \Rightarrow e : \tau' \to \tau}$$

LET' $$\frac{\Gamma \rhd e' : \tau' \quad \Gamma\{x \mapsto gen(\Gamma, \tau')\} \rhd e : \tau}{\Gamma \rhd \textbf{let val } x = e' \textbf{ in } e \textbf{ end} : \tau}$$

LETREC' $$\frac{\Gamma' \rhd e_i : \tau_i \quad \Gamma'' \rhd e : \tau}{\Gamma \rhd \textbf{let val rec } defs \textbf{ in } e \textbf{ end} : \tau}$$

$$\text{wobei} \quad defs = x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n$$
$$\Gamma' = \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}$$
$$\Gamma'' = \Gamma\{x_1 \mapsto gen(\Gamma, \tau_1), \ldots, x_n \mapsto gen(\Gamma, \tau_n)\}$$

Abb. 7.3. Modifiziertes Typinferenzsystem

Damit existiert nun für jede syntaktische Variante von *exp* genau eine Regel, mit der man den entsprechenden Typ ermitteln kann. Das bedeutet, dass man die Regeln durch die syntaktische Struktur eines Ausdrucks gesteuert von unten nach oben (und von links nach rechts) anwenden kann. Dabei erzeugt man zunächst für jeden Ausdruck, auf den man trifft und dessen Typ noch nicht bestimmt ist, eine neue Typvariable. Diese Typvariablen unterliegen nun Einschränkungen, die sich aus den Regeln des Typsystems ergeben. Solche Einschränkungen sind insbesondere die Gleichheit der Typen zweier Teilausdrücke (beispielsweise in der Regel APP). Diese Gleichheit (bei gleichzeitig maximaler Allgemeinheit) kann man durch Unifikation der entsprechenden Typausdrücke erreichen. Bevor wir also den Typinferenzalgorithmus angeben, werden wir zunächst kurz die Term-Unifikation besprechen.

Ein *Unifikator* zweier Terme ist nichts anderes als eine Substitution (s. Seite 225), die die beiden Terme „gleichmacht“ (insbesondere ist ein Typunifikator eine Substitution von Typen für Typvariablen).

Das heißt, U ist ein Unifikator für die beiden Terme τ und τ', wenn gilt: $U\tau = U\tau'$. Ein Unifikator U ist ein *allgemeinster Unifikator* (*most general unifier*) der Terme τ und τ', wenn sich jeder andere Unifikator R durch Komposition aus U und einer weiteren Substitution darstellen lässt. Das bedeutet, dass ein allgemeinster Unifikator nur für solche Variablen Definitionen enthält, die es unbedingt erfordern.

Beispiel: Ein Unifikator für die beiden Typen $\alpha \to (bool \times \gamma)$ und $int \to \beta$ ist die Substitution $[int/\alpha, string/\gamma, (bool \times string)/\beta]$, diese ist jedoch kein allgemeinster Unifikator, da sie durch Komposition der Substitution $[string/\gamma]$ mit der Substitution $[int/\alpha, (bool \times \gamma)/\beta]$ entsteht. Letztere ist ein allgemeinster Unifikator für die beiden Terme.

Wir geben nun einen Unifikationsalgorithmus in Form einer Funktion $\boldsymbol{U}$ an, die zu zwei Typausdrücken den allgemeinsten Unifikator ermittelt. Falls die Terme nicht unifizierbar sind, wird eine Fehlermeldung ausgegeben. Die folgenden Gleichungen sind sequentiell von oben nach unten anzuwenden:

$$\boldsymbol{U}(\alpha,\alpha) = []$$

$$\left.\begin{matrix}\boldsymbol{U}(\alpha,\tau)\\ \boldsymbol{U}(\tau,\alpha)\end{matrix}\right\} = \begin{cases}[\tau/\alpha] & \text{falls } \alpha \notin \tau\\ \text{Fehler} & \text{sonst}\end{cases}$$

$$\boldsymbol{U}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) = \boldsymbol{U}(U\tau_2, U\tau_4)U \quad \text{mit } U = \boldsymbol{U}(\tau_1,\tau_3)$$

$$\boldsymbol{U}(\tau,\tau') = \begin{cases}[] & \text{falls } \tau = \tau'\\ \text{Fehler} & \text{sonst}\end{cases}$$

Die erste Zeile liefert eine leere Substitution, da zwei gleiche Typvariablen bereits unifiziert sind. Der zweite Fall beschreibt die Unifikation einer Typvariablen mit einem beliebigen Typ. Diese resultiert in der angegebenen Substitution, falls die Typvariable im Typ nicht vorkommt. Diese Überprüfung nennt man *occurs check*; damit wird ausgeschlossen, dass eine Variable mit einem Typ unifiziert wird, in dem

sie selbst enthalten ist, denn es gibt keine endlichen Lösungen für solche Gleichungen. Der dritte Fall beschreibt die Unifikation von Funktionstypen: Zunächst einmal kann ein Funktionstyp nur mit einem anderen Funktionstyp unifiziert werden (Typvariablen sind ja schon abgehandelt). Dann werden zunächst beide Argumenttypen unifiziert, was zu einer Substitution U führt. Danach werden die mit U instanzierten Ergebnistypen unifiziert. Dies stellt sicher, dass die bei den Argumenttypen berechneten Einschränkungen auch in der Unifikation der Ergebnistypen berücksichtigt werden. Die resultierende Substitution wird dann mit U komponiert. Schließlich sind zwei konstante Typen (wie *int*) nur dann zu unifizieren, wenn sie gleich sind (alle anderen Möglichkeiten sind bereits betrachtet worden). In dem Fall wird keine Substitution generiert.

Als Beispiel betrachten wir die Unifikation der Typterme $\alpha \to \gamma$ und $\beta \to int \to \alpha$.

$$
\begin{aligned}
&\boldsymbol{U}(\alpha \to \gamma, \beta \to int \to \alpha) \\
&= \boldsymbol{U}(U\gamma, U(int \to \alpha))\ U \quad (\text{mit } U = \boldsymbol{U}(\alpha, \beta) = [\beta/\alpha]) \\
&= \boldsymbol{U}(\gamma, int \to \beta)\ [\beta/\alpha] \\
&= [int \to \beta/\gamma]\ [\beta/\alpha] \\
&= [int \to \beta/\gamma, \beta/\alpha]
\end{aligned}
$$

Die resultierende Substitution ergibt auf beide Terme angewandt den Typ $\beta \to int \to \beta$.

In Abb. 7.4 ist nun der Algorithmus $\boldsymbol{T}$ zur Typinferenz dargestellt. $\boldsymbol{T}$ nimmt als Parameter eine Typannahme Γ und einen Ausdruck e und liefert als Ergebnis eine Substitution U sowie den allgemeinsten Typ τ zu e. In der folgenden Beschreibung bezeichnen überstrichene Typvariablen (wie $\underline{\alpha}$) stets neue Typvariablen, die an keiner Stelle in einem Typausdruck oder in der Typannahme vorkommen. Wird eine Substitution auf eine Typannahme angewandt, so ist damit die durch Anwendung der Substitution auf jeden einzelnen Typ entstehende Typannahme gemeint. Formal liefert $\boldsymbol{T}$ zwei Ergebnisse: (1) eine Substitution und (2) einen inferierten Typ. Die Substitution wird lediglich im Algorithmus zur rekursiven Anwendung benötigt, eine Funktion zur Typprüfung wird üblicherweise nur die zweite Komponente als Ergebnis liefern.

Zur Funktionsweise des Algorithmus $\boldsymbol{T}$ sei angemerkt, dass für quantifizierte Variablen aus Typschemata jeweils neue Variablen eingesetzt werden (dies betrifft die ersten beiden Zeilen). Bei der Fallunterscheidung wird zunächst der Typ des Prädikats (τ) bestimmt. Dieser muss natürlich gleich *bool* sein, daher wird τ mit *bool* unifiziert. Man beachte, dass man nicht einfach verlangen kann, dass $\tau = bool$ sein muss, da e ja z. B. auch eine Variable sein kann, die in einem umschließenden Ausdruck definiert ist und dort (zunächst) korrekterweise einen Typ α haben kann. Deshalb muss man unifizieren und die resultierende Substitution in der weiteren Typprüfung mitführen. Danach ermittelt man den Typ von e_1, wobei man die Typannahme Γ mit den bisher ermittelten Substitutionen R und S einschränkt. Analog verfährt man mit e_2, und schließlich muss man noch die beiden für e_1 und e_2 ermittelten Typen τ_1 und τ_2 unifizieren, da beide Alternativen ja den gleichen Typ haben müssen. Das Ergebnis der Typprüfung ist dann die Gesamtheit aller Substitu-

$$\boldsymbol{T}(\Gamma, x) = ([], [\bar{\alpha}_1 / \alpha_1, \ldots, \bar{\alpha}_n / \alpha_n]\tau)$$
$$\text{wobei } \Gamma(x) = \forall \alpha_1, \ldots, \alpha_n.\tau$$

$$\boldsymbol{T}(\Gamma, c) = ([], [\bar{\alpha}_1 / \alpha_1, \ldots, \bar{\alpha}_n / \alpha_n]\tau)$$
$$\text{wobei } \Gamma(c) = \forall \alpha_1, \ldots, \alpha_n.\tau$$

$$\boldsymbol{T}(\Gamma, \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2) = (UT_1T_2SR, U\tau_1)$$
$$\text{wobei } (R, \tau) = \boldsymbol{T}(\Gamma, e)$$
$$S = \boldsymbol{U}(\tau, bool)$$
$$(T_1, \tau_1) = \boldsymbol{T}(SR\Gamma, e_1)$$
$$(T_2, \tau_2) = \boldsymbol{T}(T_1SR\Gamma, e_2)$$
$$U = \boldsymbol{U}(T_2\tau_1, \tau_2)$$

$$\boldsymbol{T}(\Gamma, e\ e') = (UTS, U\bar{\alpha})$$
$$\text{wobei } (S, \tau) = \boldsymbol{T}(\Gamma, e)$$
$$(T, \tau') = \boldsymbol{T}(S\Gamma, e')$$
$$U = \boldsymbol{U}(T\tau, \tau' \to \bar{\alpha})$$

$$\boldsymbol{T}(\Gamma, \textbf{fn } x \Rightarrow e) = (U, (U\bar{\alpha}) \to \tau)$$
$$\text{wobei } (U, \tau) = \boldsymbol{T}(\Gamma\{x \mapsto \bar{\alpha}\}, e)$$

$$\boldsymbol{T}(\Gamma, \textbf{let val } x = e' \textbf{ in } e \textbf{ end}) = (UT, \tau')$$
$$\text{wobei } (T, \tau) = \boldsymbol{T}(\Gamma, e')$$
$$(U, \tau') = \boldsymbol{T}(T\Gamma\{x \mapsto gen(T\Gamma, \tau)\}, e)$$

$$\boldsymbol{T}(\Gamma, \textbf{let val rec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n \textbf{ in } e \textbf{ end}) = (UT_nT_{n-1}\ldots T_2T_1, \tau)$$
$$\text{wobei } \Gamma' = \Gamma\{x_1 \mapsto \bar{\alpha}_1 \to \bar{\alpha}_1', \ldots, x_n \mapsto \bar{\alpha}_n \to \bar{\alpha}_n'\}$$
$$(T_1, \tau_1) = \boldsymbol{T}(\Gamma', e_1)$$
$$(T_2, \tau_2) = \boldsymbol{T}(T_1\Gamma', e_2)$$
$$\ldots$$
$$(T_n, \tau_n) = \boldsymbol{T}(T_{n-1}\ldots T_2T_1\Gamma', e_n)$$
$$\Gamma'' = T_nT_{n-1}\ldots T_2T_1\Gamma'$$
$$(U, \tau) = \boldsymbol{T}(\Gamma''\{x_1 \mapsto gen(\Gamma, \tau_1), \ldots, x_n \mapsto gen(\Gamma, \tau_n)\}, e)$$

Abb. 7.4. Typinferenz-Algorithmus

tionen sowie der Typ $U\tau_2$. (Da die bei der letzten Unifikation berechneten Einschränkungen in τ_2 noch nicht berücksichtigt sind, muss man U noch auf τ_2 applizieren.) Die übrigen Fälle kann man sich analog klarmachen. Zur Inferenz wechselseitig rekursiver Definitionen sei noch angemerkt, dass Γ' für jede zu inferierende Funktion[8] x_i bereits eine (möglichst allgemeine) Typannahme enthält, um diesen

Typ bei der Inferenz eines Ausdruck e_j zur Verfügung zu haben. Man beachte auch, dass diese Typen nicht-generisch sind und erst für die Inferenz von e generalisiert werden.

Beispielrechnungen für den Algorithmus $\boldsymbol{T}$ werden mitunter sehr aufwendig. Wir inferieren den Typ des Ausdrucks (**fn** $x \Rightarrow x$) 3 unter der Annahme $\{3 \mapsto int\}$.

$\boldsymbol{T}(\{3 \mapsto int\}, (\textbf{fn}\ x \Rightarrow x)\ 3)$
$= (UTS, U\alpha)$ mit
$(S, \tau) = \boldsymbol{T}(\{3 \mapsto int\}, \textbf{fn}\ x \Rightarrow x)$
$= (U, (U\beta) \to \tau)$ mit
$(U, \tau) = \boldsymbol{T}(\{3 \mapsto int, x \mapsto \beta\}, x) = ([], \beta)$
$= ([], \beta \to \beta)$
$(T, \tau') = \boldsymbol{T}(S\{3 \mapsto int\}, 3)$
$= \boldsymbol{T}(\{3 \mapsto int\}, 3)$
$= ([], int)$
$U = \boldsymbol{U}(T\tau, \tau' \to \alpha)$
$= \boldsymbol{U}(\beta \to \beta, int \to \alpha)$
$= \boldsymbol{U}(U\beta, U\alpha)\ U$ (mit $U = U(\beta, int) = [int/\beta]$)
$= [int/\alpha, int/\beta]$
$= ([int/\alpha, int/\beta], int)$

Man sieht an diesem Beispiel, dass Typen für Konstanten und vordefinierte Funktionen dem Algorithmus $\boldsymbol{T}$ als anfängliche Typannahmen mitgegeben werden müssen.

Selbsttestaufgabe 7.2: Erweitern Sie den Algorithmus $\boldsymbol{T}$, so dass damit auch Paare inferiert werden können. □

7.3 Implementierung durch Interpretation

Ein Programm in einer funktionalen Sprache S ist ein Ausdruck $exp \in S$, der einen Wert $v \in S'$ bezeichnet. S ist durch die Grammatikregeln für *exp* (s. Seite 220) gegeben, und die Menge S' enthält Konstanten (*con*) sowie Funktionsabschlüsse. Sei V die Menge aller Variablensymbole. Dann ist ein *Interpreter* für S eine Funktion

$$I: S \times (V \times S')^* \to S'$$

Dabei beschreibt $V \times S'$ die Menge aller Bindungen, und ein Element aus $(V \times S')^*$ ist eine Umgebung. I kann man nun einfach rekursiv über die Struktur von *exp* definieren. Im Folgenden geben wir die einzelnen Fälle an:

Konstanten sind Werte und brauchen nicht weiter ausgewertet zu werden. Daher gilt:

$$I(c, U) = c$$

[8] Wir erinnern uns, dass in einem rekursiven **let**-Ausdruck nur Funktionen definiert werden dürfen.

Eine *Variable* wird zu dem in der aktuellen Umgebung für sie gespeicherten Wert ausgewertet. Das Nachschauen dieses Wertes geschieht über eine Funktion *lookup*, die man sich wie folgt definiert vorstellen kann:

$$lookup(x, (y, v)\cdot U) = \begin{cases} v & \text{falls } x = y \\ lookup(x, U) & \text{sonst} \end{cases}$$

Der Fall, dass eine Variable in der Umgebung nicht gefunden wird, bleibt undefiniert. Wir gehen davon aus, dass dies in früheren Phasen der Übersetzung bereits überprüft worden ist. Die Definition von *I* lautet nun einfach:

$$I(x, U) = lookup(x, U)$$

Man beachte, dass der mit *lookup* ermittelte Wert nicht weiter ausgewertet zu werden braucht, da aufgrund der call-by-value-Semantik des Interpreters Ausdrücke nur ausgewertet als Parameter übergeben werden und dementsprechend nur Werte in der Umgebung gespeichert werden (siehe Gleichungen für Abstraktion und **let**-Ausdruck).

Die Auswertung der *Fallunterscheidung* verläuft ganz wie erwartet: Zunächst ermittelt man den Wert der Bedingung *e* und berechnet in Abhängigkeit von dem Ergebnis entweder den Ausdruck der ersten oder der zweiten Alternative:

$$I(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2, U) = \begin{cases} I(e_1, U) & \text{falls } I(e, U) = true \\ I(e_2, U) & \text{falls } I(e, U) = false \end{cases}$$

Bei der Auswertung einer *Applikation* muss man drei Fälle unterscheiden: Die Applikation von unären oder binären vordefinierten Funktionen sowie die Applikation von Abstraktionen. In den ersten beiden Fällen wird die Applikation über eine nicht näher betrachtete Funktion *apply* realisiert. Dabei ist nur zu beachten, dass die Argumente per call-by-value übergeben werden:

$$I(c\ e, U) = apply(c, I(e, U))$$
$$I(e_1\ c\ e_2, U) = apply(c, (I(e_1, U), I(e_2, U)))$$

Bei der Applikation einer Abstraktion muss man diese zunächst zu einer closure (**fn** $x \Rightarrow e, U$') auswerten und den definierenden Funktionsausdruck *e* auf den Wert des Argumentes applizieren. Dabei ist zu beachten, dass die Auswertung in der Umgebung der closure, *U*', erfolgt, um die statische Bindung freier Variablen zu garantieren (s. Seite 218).

$$I(e_1\ e_2, U) = I(e, (x, v)\cdot U')$$
$$\text{wobei } I(e_1, U) = (\textbf{fn } x \Rightarrow e, U')$$
$$\text{und } I(e_2, U) = v$$

Durch die Auswertung von e_2 vor der Bindung an *x* wird die call-by-value-Parameterübergabe realisiert.

Eine *Abstraktion* wird zu einer closure ausgewertet. Da closures Werte sind, brauchen sie nicht weiter ausgewertet zu werden.

$$I(\textbf{fn}\ x \Rightarrow e, U) = (\textbf{fn}\ x \Rightarrow e, U)$$

Die Interpretation für einen **let**-*Ausdruck* **let val** $x = e'$ **in** e **end** erweitert die aktuelle Umgebung um die Bindung (x, v), wobei v den Wert von e' bezeichnet. In dieser erweiterten Umgebung wird dann der Ausdruck e berechnet.

$$I(\textbf{let val}\ x = e'\ \textbf{in}\ e\ \textbf{end}, U) = I(e, (x, v)\cdot U)$$
$$\text{wobei } I(e', U) = v$$

Eine *rekursiver* **let**-*Ausdruck* wird im Prinzip ganz ähnlich interpretiert wie ein nicht-rekursiver. Es gibt jedoch zwei Unterschiede: Zum einen müssen wir anstelle einer einzigen Definition eine ganze Liste von Definitionen in die aktuelle Umgebung einfügen – dies bereitet wohl kaum Probleme. Zum anderen aber, und dies ist schon wesentlich komplizierter zu bewerkstelligen, können die definierenden Ausdrücke die definierten Variablen enthalten. Betrachten wir als Beispiel die obige Definition von *fak*:

```
let   val rec fak = fn x ⇒ if x<3 then x else x*fak (x–1)
in
      fak 3
end
```

Angenommen, der gesamte Ausdruck ist in einer Umgebung U auszuwerten, so muss die Auswertung des Ausdrucks *fak* 3 in der Umgebung $U' = (fak, f)\cdot U$ erfolgen, wobei f das Ergebnis der Auswertung des Funktionsausdrucks **fn** $x \Rightarrow$ **if** x<3 **then** x **else** x**fak* $(x–1)$ ist (wir bezeichnen diesen im Folgenden mit F). Dessen Auswertung muss jedoch ebenfalls in U' erfolgen, da in ihm die Variable *fak* verwendet wird. Stellt diese zirkuläre Definition nicht einen unauflösbaren Widerspruch dar? Nein. Denn wir wissen bereits von der Interpretation von Abstraktionen, dass f eine closure sein muss, bestehend aus dem Funktionsausdruck F und einer Umgebung. Diese Umgebung muss nun gerade U' sein, um die rekursive Referenz auf *fak* zu ermöglichen. Wir erhalten also eine zirkuläre Datenstruktur, die man wie folgt konstruieren kann: Zunächst verwendet man einen Platzhalter Ω anstelle von U' in der Bindung für *fak*, d. h., man bildet $U' = (fak, (F, \Omega))\cdot U$. Danach überschreibt man dann Ω mit U'. Dies ist noch einmal in Abb. 7.5 dargestellt.

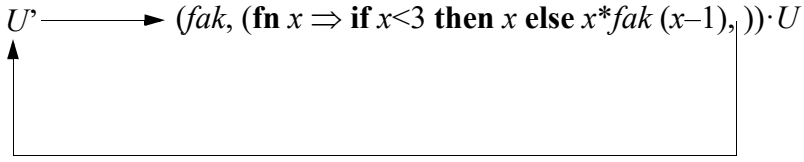

Abb. 7.5. Zyklische Umgebung

Damit wird erreicht, dass bei der Auswertung der Applikation *fak* 3 der Ausdruck **if** $x<3$ **then** x **else** x**fak* $(x-1)$ in der Umgebung $(x, 3)\cdot U'$ berechnet wird. Dadurch wird beim rekursiven Aufruf die korrekte closure für *fak* gefunden.

Für die Interpretation eines rekursiven **let**-Ausdrucks mit n Definitionen erhalten wir entsprechend die folgende Interpretation:

$$
\begin{aligned}
&I(\textbf{let val rec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n \textbf{ in } e \textbf{ end}, U) = I(e, U') \\
&\quad \text{wobei} \quad U' = (x_1, v_1)\cdot\ldots\cdot(x_n, v_n)\cdot U \\
&\quad \text{und} \quad v_1 = (e_1, U') \\
&\qquad\qquad \ldots \\
&\qquad\qquad v_n = (e_n, U')
\end{aligned}
$$

Ein entscheidendes Element funktionaler Sprachen ist zweifelsohne die Rekursion. In der Beschreibung des Interpreters haben wir gleichermaßen von Rekursion intensiven Gebrauch gemacht. Es gibt allerdings nur wenige Prozessorarchitekturen, die Rekursion elementar zur Verfügung stellen. Unter diesem Gesichtspunkt sind wir von einer realistischen Implementierung doch noch ein gutes Stück entfernt. Deshalb betrachten wir im Folgenden die Übersetzung der funktionalen Kernsprache in eine Maschinensprache. Allerdings beziehen wir uns nicht auf einen konkreten Prozessor, sondern auf eine abstrakte Maschine, die SECD-Maschine.

7.4 Implementierung durch Übersetzung

Wir beschreiben in Abschnitt 7.4.1 die Architektur der SECD-Maschine sowie die Maschinenbefehle und deren Wirkung. Dies erfolgt zunächst in einer vereinfachten Darstellung, in der wir die Behandlung von Rekursion ignorieren. Die Übersetzung von ML in SECD-Code ist dann Gegenstand des Abschnitts 7.4.2. Die Realisierung von Rekursion folgt schließlich in Abschnitt 7.4.3.

7.4.1 Die SECD-Maschine

Die SECD-Maschine arbeitet mit vier Registern.[9] Die Bezeichnungen der Register geben der SECD-Maschine ihren Namen:

S *Stack*. Der Stack dient zur Ablage von Zwischenergebnissen bei der Berechnung von Ausdrücken.

E *Environment*. Im Environment (Umgebung) werden Variablenbindungen verwaltet, die zur Auswertung von Ausdrücken benötigt werden.

C *Control*. Dies ist der eigentliche Programmspeicher.

---

[9] Dies sind nicht Register im Sinne herkömmlicher Prozessorarchitekturen, vielmehr kann ein SECD-Register beliebig komplexe Werte enthalten.

D *Dump*. Dieses Register dient zum Zwischenspeichern kompletter Maschinenkonfigurationen. Dies ist immer bei einer Funktionsapplikation erforderlich: In der SECD-Maschine werden Funktionen als closures repräsentiert, wobei eine closure aus Maschinencode und einer Umgebung besteht, in der dieser Code ausgeführt werden soll. Um die closure applizieren zu können, muss man den Code in den Programmspeicher bringen und das Environment installieren. Die alten Zustände dieser Register werden dann auf den Dump gesichert und nach Abarbeitung der Applikation reinstalliert.

Der Vorrat an Maschinenbefehlen der SECD-Maschine lautet:

LD	Laden einer Konstanten
LDV	Laden einer Variablen
LDC	Laden einer closure
APP	Applizieren einer closure
RAP	Rekursives Applizieren
DUM	Erzeugen von „Dummy“-Einträgen im Environment
COND	Bedingter Sprung zu einer Befehlsfolge
RET	Rückkehr nach bedingtem Sprung
ADD *NOT* *EQ* ...	} vordefinierte Funktionen

Die SECD-Maschine kann man sich als Automaten vorstellen, dessen aktueller Zustand jeweils durch ein Viertupel (S, E, C, D) gegeben ist. Die Funktionsweise der SECD-Maschine ist dann durch *Transitionsregeln* der folgenden Form gegeben:

$$(S, E, C, D) \mapsto (S', E', C', D')$$

Für jeden Befehl werden wir die Transitionsregel anschließend beschreiben. In der Tat sind alle Register Stacks, die in den Transitionsregeln mit der bereits verwandten Listenschreibweise notiert werden, d. h., einen Stack mit oberstem Element x und Rest-Stack s schreiben wir als $x{\cdot}s$.

Die Ladebefehle (*LD*, *LDV* und *LDC*) bringen allesamt ihr jeweiliges Argument auf den Stack. Die einfachste Ladeoperation ist *LD*:

$$(S, E, LD\ x{\cdot}C, D) \mapsto (x{\cdot}S, E, C, D)$$

Beim Laden einer Variablen muss deren Wert im Environment nachgesehen werden. Dies geschieht in etwas anderer Weise als in der Interpreter-Implementierung: Eine Umgebung wird in der SECD-Maschine nicht als Liste von Bindungen dargestellt, sondern einfach als Liste von Werten. Die Position des zu einer Variablen

gehörenden Wertes wird bereits bei der Übersetzung des funktionalen Programms berechnet und erscheint als Argument des *LDV*-Befehls.

$$(S, x_1 \cdot \ldots \cdot x_n \cdot E, LDV\ n \cdot C, D) \mapsto (x_n \cdot S, x_1 \cdot \ldots \cdot x_n \cdot E, C, D)$$

Der *LDC*-Befehl hat als Parameter eine Liste von Maschinenbefehlen, die der Übersetzung einer Funktion entsprechen. Aus dieser Liste und der aktuellen Umgebung *E* wird bei Abarbeitung des *LDC*-Befehls eine closure erzeugt, die auf den Stack geladen wird. Eine closure in der SECD-Maschine ist ein Paar bestehend aus einer Liste von Maschinenbefehlen und einer Liste von Werten.

$$(S, E, (LDC\ C') \cdot C, D) \mapsto ((C', E) \cdot S, E, C, D)$$

Bei der Bearbeitung des *APP*-Befehls wird auf dem Stack eine closure (*C*', *E*') erwartet sowie das Argument *x*, auf das die closure appliziert wird. Im Weiteren soll dann der Code *C*' in dem um *x* erweiterten Environment *E*' ausgeführt werden, d. h., die Übersetzung der durch *C*' repräsentierten Funktion erfolgt so, dass Referenzen auf das Funktionsargument als Zugriffe auf die erste Variable des Environments (mit *LDV* 1) dargestellt werden und die Nummerierung der globalen Variablen bei 2 beginnt. Die Ausführung der Applikation erreicht man nun durch Installation von *C*' im Programmspeicher und von $x \cdot E$' als neuer Umgebung. Ist *C*' vollständig abgearbeitet, wird der in *C* auf *APP* folgende Befehl ausgeführt, und zwar in der alten Umgebung. Dies bedeutet, dass sowohl der restliche Programmspeicher als auch die aktuelle Umgebung vor Auswertung der closure gesichert werden müssen. Dazu wird der Dump verwendet: Dort wird der gesamte Maschinenzustand gespeichert, der nach Beendigung der closure-Bearbeitung wiederherzustellen ist.

$$((C', E') \cdot x \cdot S, E, APP \cdot C, D) \mapsto ([], x \cdot E', C', (S, E, C) \cdot D)$$

Das Laden des closure-Codes in den Programmspeicher entspricht einem Sprung in konventionellen Maschinen. Wir müssen noch erklären, wie nach Abarbeitung des Codes der entsprechende „Rücksprung" erfolgt. Das Ende des Funktionscodes erkennt man ganz einfach an einem leeren Programmspeicher. Dann steht das Resultat der entsprechenden Applikation (*x*) oben auf dem Stack. Nun wird der auf dem Dump gesicherte Maschinenzustand wiederhergestellt, wobei *x* nicht verworfen, sondern oben auf den gesicherten Stack gepackt wird.

$$(x \cdot S', E', [], (S, E, C) \cdot D) \mapsto (x \cdot S, E, C, D)$$

Bei der Applikation von Befehlen für vordefinierte Funktionen (zumindest bei den zweistelligen) ist zu beachten, dass die Argumente in umgekehrter Reihenfolge auf dem Stack liegen. Dies ergibt sich aus der Übersetzung von Programmen in Maschinencode (s. Abschnitt 7.4.2). Ansonsten sind die Transitionen offensichtlich:

$$(y \cdot x \cdot S, E, ADD \cdot C, D) \mapsto (x+y \cdot S, E, C, D)$$
$$(y \cdot x \cdot S, E, EQ \cdot C, D) \mapsto (x=y \cdot S, E, C, D)$$
$$\ldots$$

Der *COND*-Befehl realisiert einen bedingten Sprung. Auf dem Stack wird ein boolescher Wert erwartet, der die Auswahl einer von zwei Codesequenzen bestimmt.

Nach Ausführung einer der beiden Alternativen wird mit dem nächsten Befehl im Programmspeicher fortgefahren. Deshalb muss der Code auf dem Dump gesichert werden. Stack und Environment bleiben prinzipiell erhalten und brauchen daher nicht gesichert zu werden. Wir übergeben also zwei leere Listen.

$$(\mathit{true}\cdot S, E, \mathit{COND}(C_1, C_2)\cdot C, D) \mapsto (S, E, C_1, ([], [], C)\cdot D)$$
$$(\mathit{false}\cdot S, E, \mathit{COND}(C_1, C_2)\cdot C, D) \mapsto (S, E, C_2, ([], [], C)\cdot D)$$

Der Rücksprung aus einer *COND*-Verzweigung erfolgt durch den Befehl *RET* (durch die Übersetzung wird sichergestellt, dass jede der beiden Alternativen C_1 und C_2 als letzten Befehl *RET* enthält). Daher kann man die Fallunterscheidung von der Applikation unterscheiden und beim Rücksprung lediglich den Code reinstallieren und die leeren Listen für Stack und Environment ignorieren.

$$(S, E, [\mathit{RET}],([], [], C)\cdot D) \mapsto (S, E, C, D)$$

Bei der Ausführung eines *RAP*-Befehls ergibt sich ein ähnliches Zirkularitätsproblem wie schon bei der Interpretation des rekursiven **let**-Ausdrucks in Abschnitt 7.3. Um die Arbeitsweise der SECD-Maschine bei diesem Befehl (und auch beim Befehl *DUM*) genau beschreiben zu können, müssen wir den Zusammenhang zwischen einem rekursiven **let**-Ausdruck und dem *RAP*-Befehl verstehen, und dies erfordert Kenntnisse über die Übersetzung in Maschinensprache. Deshalb schieben wir die Behandlung von Rekursion auf, bis wir im nächsten Abschnitt die Übersetzung näher betrachtet haben.

Ein Maschinenprogramm *C* wird nun von der SECD-Maschine wie folgt bearbeitet: Beginnend mit dem Zustand ([], [], *C*, []), werden die Transitionsregeln so lange angewandt, bis Programmspeicher und Dump leer sind. Das Ergebnis befindet sich dann auf dem Stack.

Selbsttestaufgabe 7.3: Das folgende Maschinenprogramm beschreibt die Anwendung der Successor-Funktion auf die Zahl 3:

[*LD* 3, *LDC* [*LDV* 1, *LD* 1, *ADD*], *APP*]

Vollziehen Sie die Bearbeitung des Programms durch die SECD-Maschine nach, d. h., geben Sie die entsprechende Folge von Maschinenzuständen an. □

7.4.2 Übersetzung von ML in SECD-Code

Wir hatten in der Erklärung zum *LDV*-Befehl bereits erwähnt, dass im Environment lediglich Werte (ohne Variablennamen) stehen und dass der Zugriff über die Position des Wertes in der Environment-Liste erfolgt. Diese Positionen werden vom Compiler berechnet. Daher müssen die Übersetzungsregeln relativ zu einer Liste von Variablennamen formuliert werden, die der Liste von Werten im Environment entspricht. Dagegen benötigen wir das Environment selbst (d. h. die Werte) bei der Übersetzung nicht. Die Namensliste wird nun während der Übersetzung z. B. durch Abstraktion oder **let**-Ausdruck erweitert, ganz so, wie es mit dem Environment zur Laufzeit geschieht. Bei der Übersetzung eines Variablennamens wird dann die Posi-

tion des Namens in der Liste ermittelt, und dies ergibt den Parameter des *LDV*-Befehls. Da die Namensliste während der Übersetzungsphase genauso auf- und abgebaut wird wie das Environment zur Laufzeit der SECD-Maschine, ist gewährleistet, dass die für Variablen ermittelten Positionswerte stets auf den richtigen Wert im Environment verweisen.

Im Folgenden beschreiben wir für jeden Mini-ML-Ausdruck die Übersetzung in eine Liste von SECD-Maschinenbefehlen. Dazu definieren wir eine Funktion $\ddot{U}$, die als zweites Argument eine Liste N von Variablennamen hat.

Konstanten werden direkt in Ladebefehle übersetzt:

$$\ddot{U}(c, N) = [LD\ c]$$

Für *Variablen* werden ebenfalls Ladebefehle generiert, wobei die Funktion *position* die Stelle der Variablen x in der Namensliste N ermittelt:

$$position(x, y{\cdot}N) = \begin{cases} 1 & \text{falls } x = y \\ 1 + position(x, N) & \text{sonst} \end{cases}$$

Die Übersetzung von Variablen lautet nun:

$$\ddot{U}(x, N) = [LDV\ position(x, N)]$$

Die Übersetzung von Variablen ist im Übrigen die einzige Stelle, an der in der Namensliste N gesucht wird.

Die *Fallunterscheidung* wird in einen *COND*-Befehl übersetzt. Da dieser den booleschen Wert der Bedingung auf dem Stack erwartet, wird die Übersetzung der Bedingung dem *COND*-Befehl vorangestellt. Das Argument des *COND*-Befehls ist ein Paar bestehend aus den Übersetzungen der beiden Alternativen, jeweils gefolgt von einem *RET*-Befehl.

$$\ddot{U}(\textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2, N) = \ddot{U}(c, N){\cdot}[COND(\ddot{U}(e_1, N){\cdot}[RET], \ddot{U}(e_2, N){\cdot}[RET])]$$

Wir bezeichnen hier die Konkatenation von Listen ebenfalls mit „·“.

Betrachten wir nun die Übersetzung von *Applikationen*. Bei vordefinierten Funktionen wird zunächst Code für die Argumente erzeugt. Daran wird ein Maschinenbefehl angehängt, der die vordefinierte Funktion realisiert. Die Korrespondenz zwischen vordefinierten Funktionen und Maschinenbefehlen ist über eine Funktion *code* beschrieben (also $code(+) = [ADD]$, $code(-) = [SUB]$, …).

$$\ddot{U}(c\ e, N) = \ddot{U}(e, N){\cdot}code(c)$$
$$\ddot{U}(e_1\ c\ e_2, N) = \ddot{U}(e_1, N){\cdot}\ddot{U}(e_2, N){\cdot}code(c)$$

An der Übersetzung für binäre Funktionen kann man nun auch erkennen, warum binäre Maschinenbefehle ihre Argumente in umgekehrter Reihenfolge auf dem Stack vorfinden. Denn die Übersetzung des Ausdrucks 4–1 ergibt offensichtlich die Befehlsfolge

[*LD* 4, *LD* 1, *SUB*]

Die Verarbeitung dieser Sequenz durch die SECD-Maschine bringt zunächst die 4 und danach die 1 auf den Stack, d. h., der Befehl *SUB* findet sein zweites Argument über dem ersten auf dem Stack.

Die Applikation einer Abstraktion wird über den *APP*-Befehl realisiert. Dieser erwartet oben auf dem Stack eine closure und direkt darunter das Argument. Daher wird die Abstraktion als erste übersetzt, gefolgt von dem zu applizierenden Ausdruck und dem *APP*-Befehl.

$$\ddot{U}(e\ e', N) = \ddot{U}(e', N) \cdot \ddot{U}(e, N) \cdot [APP]$$

Eine *Abstraktion* (**fn** $x \Rightarrow e$) wird in einen *LDC*-Befehl übersetzt, dessen Argument die aus der Übersetzung des Funktionsrumpfes e resultierende Befehlsfolge ist. Zu beachten ist, dass e relativ zu der um x erweiterten Namensliste übersetzt wird.

$$\ddot{U}(\textbf{fn}\ x \Rightarrow e, N) = [LDC\ \ddot{U}(e, x \cdot N)]$$

Die Übersetzung eines **let**-Ausdrucks ergibt sich direkt aus der Korrespondenz des Ausdrucks **let val** $x = e'$ **in** e **end** zum Ausdruck (**fn** $x \Rightarrow e$) e'.

$$\ddot{U}(\textbf{let val}\ x = e'\ \textbf{in}\ e\ \textbf{end}, N) = \ddot{U}(e', N) \cdot [LDC\ \ddot{U}(e, x \cdot N)] \cdot [APP]$$

Die Übersetzung eines rekursiven **let**-Ausdrucks betrachten wir zusammen mit den noch ausstehenden Transitionsregeln im nächsten Abschnitt.

Selbsttestaufgabe 7.4: Betrachten Sie die Funktion *twice*, die durch den folgenden Ausdruck definiert ist:

$$\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x)$$

Übersetzen Sie die Funktion *twice* in SECD-Code. □

7.4.3 Behandlung von Rekursion

Wir untersuchen zunächst die Übersetzung eines rekursiven **let**-Ausdrucks und beschreiben anschließend die dazugehörigen Transitionsregeln der SECD-Maschine.

Der Unterschied zwischen einem rekursiven **let**-Ausdruck und einem nicht-rekursiven ist zum einen, dass mehrere Definitionen vorgenommen werden können und zum anderen, dass die definierenden Ausdrücke die definierten Variablen enthalten können. Entsprechend wollen wir die Übersetzung in zwei Schritten erklären: Zunächst betrachten wir die Erweiterung eines **let**-Ausdrucks auf mehrere definierte Variablen. Danach behandeln wir die Rekursion.

Die Übersetzung eines Ausdrucks **let val** $x = e'$ **in** e **end** erzeugt zunächst Code für e' (z. B. C'), gefolgt vom Code für e (z. B. *LDC C*) und einem *APP*-Befehl. Zur Laufzeit bewirkt der Code C', dass der Wert von e' (z. B. v) oben auf dem Stack erscheint, *LDC C* lädt eine closure darüber, und *APP* veranlasst die Ausführung von C im um v erweiterten Environment. Betrachten wir nun die erweiterte Form **let val**

$x_1 = e_1$ **and** ... **and** $x_n = e_n$ **in** e **end**. Wir müssen sowohl die Übersetzung anpassen als auch die SECD-Transitionsregel für *APP*. Die Übersetzung muss zunächst Code für alle Definitionen erzeugen, d. h. die konkatenierte Folge der Befehle von C_1, …, C_n, wobei C_i der Code für den Ausdruck e_i ist. Anschließend wird der *LDC*-Befehl mit dem Code für e generiert. Dabei ist zu beachten, dass die Übersetzung von e relativ zu einer um $[x_1, ..., x_n]$ erweiterten Namensliste erfolgt. Darüber hinaus muss der abschließende *APP*-Befehl Kenntnis über die Anzahl n der auf dem Stack liegenden Definitionen haben, um genau diese auf das Environment laden zu können. Daher wird der *APP*-Befehl mit einem entsprechenden Parameter ausgestattet. Die Übersetzung lautet nun:

$$\begin{aligned}&\ddot{U}(\textbf{let val } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n \textbf{ in } e \textbf{ end}, N) = \\ &\quad \ddot{U}(e_1, N)\cdot\ldots\cdot\ddot{U}(e_n, N)\cdot[LDC\ \ddot{U}(e, [x_1, ..., x_n]\cdot N), APP\ n]\end{aligned}$$

Die Transitionsregel für *APP* muss nun n Werte auf das Environment der zu applizierenden closure laden. Zu beachten ist dabei wiederum, dass die Werte in gespiegelter Reihenfolge auf dem Stack erscheinen.

$$((C', E')\cdot[x_n, ..., x_1]\cdot S, E, APP\ n\cdot C, D) \mapsto ([], [x_1, ..., x_n]\cdot E', C', (S, E, C)\cdot D)$$

Wenden wir uns der Rekursion in den Ausdrücken e_1, …, e_n zu. Um Verweise auf irgendwelche x_j korrekt zu behandeln, muss die Übersetzung eines jeden Ausdrucks e_i in der um x_1, ..., x_n erweitereten Namensliste erfolgen. An die Übersetzung wird ein *RAP*-Befehl angefügt, dessen Transitionsregel sich von der des *APP*-Befehls deutlich unterscheidet. Da die e_i in einem rekursiven **let**-Ausdruck Funktionen sein müssen, werden diese in *LDC*-Befehle übersetzt. Zur Laufzeit wird dann für jeden *LDC*-Befehl eine closure $cl_i = (C_i, E')$ gebildet, die auf den Stack geladen wird. Das Environment E' einer jeden solchen closure muss aber das Environment sein, das bei Bearbeitung des *RAP*-Befehls durch Laden der closures in das aktuelle Environment erst entsteht, denn in den Codesequenzen C_i können ja Ladebefehle für beliebige Werte (d. h. also für beliebige cl_i) vorkommen. Hier ergibt sich eine ähnlich zirkuläre Situation wie im Interpreter. Über Gleichungen lässt sich der Zustand (S', E', C, D) der SECD-Maschine nach Laden aller closures auf den Stack so beschreiben:

$$\begin{aligned}S' &= [(C_n, E'), \ldots, (C_1, E')]\cdot S \\ E' &= [(C_1, E'), \ldots, (C_n, E')]\cdot E\end{aligned}$$

Wie kann man dieses Verhalten nun realisieren? Wir bemerken, dass die closures cl_i zunächst (d. h. bis zur Ausführung des *RAP*-Befehls) lediglich auf den Stack geladen und nicht appliziert werden. Daher wird auf das jeweils enthaltene Environment E' auch noch nicht zugegriffen. Es reicht somit aus, in E' für die Werte e_i zunächst nur Platzhalter vorzusehen, in die dann nach Bearbeitung der letzten Definition alle Werte eingetragen werden können. Dies bedeutet in der Tat ein nachträgliches, imperatives Überschreiben im Environment E'. Eine Liste von n Platzhaltern wird durch den Befehl *DUM n* vorne an das aktuelle Environment angefügt. Da dieses erweiterte Environment von allen closures cl_i benötigt wird, erscheint der *DUM*-Befehl vor der Übersetzung der Ausdrücke e_i. Damit wird ein rekursiver **let**-Ausdruck also wie folgt übersetzt. Es sei $N' = [x_1, ..., x_n]\cdot N$:

$$\ddot{U}(\textbf{let val rec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n \textbf{ in } e \textbf{ end}, N) = \\ DUM\ n{\cdot}\ddot{U}(e_1, N'){\cdot}\ldots{\cdot}\ddot{U}(e_n, N'){\cdot}[LDC\ \ddot{U}(e, N'), RAP\ n]$$

Betrachten wir nun die Verarbeitung durch die SECD-Maschine. Zunächst erzeugt der *DUM*-Befehl ein Environment mit *n* Platzhaltern an der Spitze. Diese Platzhalter können wir uns als Zeiger auf einen undefinierten Wert vorstellen.

$$(S, E, DUM\ n{\cdot}C, D) \mapsto (S, E', C, D)$$
$$\text{mit } E' := \underbrace{[\Omega, \ldots, \Omega]}_{n\text{–mal}} \cdot E$$

Der Bezeichner *E*' wird hier im Sinne einer imperativen Variablen verwandt: Der durch das Überschreiben der Platzhalter Ω bewirkte Seiteneffekt wird dann an allen Stellen sichtbar, an denen auf *E*' verwiesen wird.

Die folgenden *LDC*-Befehle laden nun die closures cl_i auf den Stack, die jeweils das unfertige Environment *E*' enthalten. Danach wird die für *e* übersetzte closure (*C*, *E*') (ebenfalls mit Zeiger auf Environment *E*') auf den Stack geladen. Der *RAP*-Befehl appliziert nun die closure (*C*, *E*') auf die im Stack darunter liegenden closures cl_i. Das erforderliche Laden dieser closures in das Environment *E*' erfolgt durch Überschreiben der darin enthaltenen Platzhalter.

Der Zustand vor Ausführung des *RAP*-Befehls ist also:

$$([(C, E'), (C_n, E'), \ldots, (C_1, E')]{\cdot}S, E', RAP\ n{\cdot}C', D)$$

Nun werden die folgenden Zuweisungen an die Platzhalter in *E*' ausgeführt:

$$E'[1] := (C_1, E'); \ldots; E'[n] := (C_n, E')$$

Das heißt, wir erhalten für *E*':

$$E' = [(C_1, E'), \ldots, (C_n, E')]{\cdot}E$$

Dann ergibt sich der folgende Zustand nach Ausführung des *RAP*-Befehls:

$$([], E', C, (S, E, C'){\cdot}D)$$

Selbsttestaufgabe 7.5: Übersetzen Sie den folgenden Ausdruck, der die Fakultätsfunktion repräsentiert, in SECD-Code:

let val rec *fak* = **fn** $x \Rightarrow$ **if** x<3 **then** x **else** x**fak* (x–1) **in** *fak* **end** □

7.5 Literaturhinweise

Eine äußerst empfehlenswerte Einführung in ML bietet Ullman (1998). Die vielen z. T. sehr ausführlich erklärten Beispiele sind besonders für Anfänger hilfreich, ebenso wie die Erläuterungen zu häufig gemachten Fehlern. Das Buch basiert auf

dem ML-Standard, der in (Milner et al. 1997) definiert worden ist. Eine sehr gute, allgemeine Einführung in die funktionale Programmierung gibt das Buch von Bird (1998), das auf der Sprache Haskell basiert. Einen guten Überblick über funktionale Programmiersprachen verschafft der Übersichtsartikel von Hudak (1989), und Plädoyers für funktionale Programmierung und Vergleiche mit beispielsweise imperativen Sprachen finden sich u. a. in den beiden Artikeln (Hughes 1989) und (Backus 1978). Als Lehrbücher, die sowohl eine Einführung in die funktionale Programmierung auf der Basis von ML geben als auch Übersetzungsaspekte wie die Typinferenz und die SECD-Maschine beschreiben, sind (Reade 1989) und (Erwig 1999) zu nennen, wobei jedoch im Buch von Reade im Unterschied zu (Ullman 1998) und (Erwig 1999) ein älterer ML-Standard verwandt wird. In (Erwig 1999) wird auch eine vollständige Implementierungen des hier vorgestellten Interpreters, der SECD-Maschine sowie des SECD-Compilers angegeben. Fortgeschrittene Beispiele für ML-Programmierung aus dem Bereich Datenstrukturen bietet das äußerst empfehlenswerte Buch von Okasaki (1999). Dort wird insbesondere auf den Unterschied zu Datenstrukturen in imperativen Sprachen eingegangen.

Die vorgestellte Technik zur Typinferenz ist für die meisten typisierten funktionalen Sprachen gleich. Die polymorphen Typsysteme moderner funktionaler Programmiersprachen basieren überwiegend auf den Arbeiten von Milner und Damas (Milner 1978, Damas und Milner 1982). Der für die Typinferenz benötigte Unifikationsalgorithmus stammt von Robinson (1965). Eine gute Einführung in Typsysteme und Typinferenz gibt auch der Artikel von Cardelli (1987). Ausführliche Beispiele für die Anwendung des Typinferenzalgorithmus findet man im Buch von Field und Harrison (1988). Allerdings basiert die Darstellung (wie auch die in (Cardelli 1987)) auf einer etwas anderen Kernsprache, die anstelle eines rekursiven **let** einen Fixpunktoperator verwendet. Reade (1989) beschreibt die Typinferenz über eine Implementierung in ML. Erweiterungen des Typsystems um kontrolliertes Overloading wie es von der Sprache Haskell angeboten wird, sind u. a. in (Wadler und Blott 1989, Jones 1995) beschrieben. Für entsprechende Erweiterungen des Typinferenzalgorithmus gibt es verschiedene Ansätze (Jones 1992, Nipkow und Snelting 1991, Nipkow und Prehofer 1993, Hall et al. 1996). Mehr über Typsysteme und Polymorphismus kann man u. a. im Übersichtsartikel von Mitchell (1990) finden. Dort wird insbesondere auch auf die Semantik von Typen eingegangen.

Die Idee der SECD-Maschine stammt von Landin (1964). Eine sehr ausführliche Beschreibung der SECD-Maschine bietet (Henderson 1980). Wie schon erwähnt, findet man Beschreibungen der SECD-Maschine auch in (Reade 1989) und (Erwig 1999). Eine andere abstrakte Maschine, die zur Implementierung von ML genutzt wird, ist die Categorical Abstract Machine (CAM). Sie ist u. a. im Buch von Field und Harrison (1988) beschrieben; dieses Buch befasst sich im übrigen sehr ausführlich mit Implementierungsaspekten, insbesondere werden eine ganze Reihe verschiedener Optimierungsverfahren untersucht. Einen kurzen Vergleich der CAM mit der SECD-Maschine findet man auch im Buch von Reade.

Speziell für ML gibt es Implementierungsansätze, die sogenannte *continuations* (Fortsetzungen) als Zwischensprache benutzen und darauf viele Optimierungen

durchführen können. Dies ist sehr ausführlich in (Appel 1992) dargestellt. Ein Übersetzer von ML nach C ist in (Tarditi, Acharya und Lee 1990) beschrieben, und einen Baukasten für ML-Implementierungen bietet das ML-Kit (Birkedal et al.1993). Moderne Übersetzungsverfahren für ML basieren z. T. auf typisierten Zwischensprachen (Shao 1997b) und sind in (Shao und Appel 1995, Tarditi et al. 1996) oder auch (Shao 1997a) beschrieben.

Die Implementierung von sogenannten „lazy“ Sprachen wie Miranda oder Haskell basiert zumeist auf der sogenannten *Graph-Reduktion*: Ausdrücke werden als Graphen repräsentiert, wobei deren Auswertung durch Transformationsregeln auf Graphen definiert ist. Graph-Reduktion wird im Buch von Field und Harrison relativ ausführlich beschrieben. Die Graph-Reduktion wird auch von Wilhelm und Maurer (2007) behandelt. Darüber hinaus findet man in (Peyton Jones 1987) und (Peyton Jones und Lester 1992) eine detaillierte und umfassende Darstellung der Implementierung einer Sprache wie Miranda.

Lösungen zu den Selbsttestaufgaben

Aufgabe 7.1

(a) Wir haben bereits gesehen, wie man für den Ausdruck **fn** $x \Rightarrow x$ den Typ $\alpha \to \alpha$ herleitet. Analog kann man für den Ausdruck **fn** $x \Rightarrow 1$ den Typ $\alpha \to \mathit{int}$ herleiten. Durch Generalisierung erhält man für **fn** $x \Rightarrow x$ das Typschema $\forall\alpha.\alpha \to \alpha$, aus dem man mit der Regel SPEC den Typ $(\alpha \to \mathit{int}) \to (\alpha \to \mathit{int})$ erhält. Nun kann man die Regel APP anwenden und erhält für die Applikation (**fn** $x \Rightarrow x$) (**fn** $x \Rightarrow 1$) den Typ $\alpha \to \mathit{int}$.

(b) Mit $\Gamma' = \{x \mapsto \alpha, y \mapsto \beta\}$ erhält man durch zweimaliges Anwenden der Regel VAR für x und für y: $\Gamma' \rhd x : \alpha$ und $\Gamma' \rhd y : \beta$. Mit der Regel ABS bekommt man so den Typ $\beta \to \alpha$ für die Abstraktion **fn** $y \Rightarrow x$ unter der Annahme $\Gamma = \{x \mapsto \alpha\}$. Durch erneute Anwendung der Regel ABS erhält man $\{\} \rhd$ **fn** $x \Rightarrow$ **fn** $y \Rightarrow x : \alpha \to \beta \to \alpha$. Nun kann man in diesem Typ α generalisieren und sofort wieder zu *int* spezialisieren, so dass wir den Typ $\mathit{int} \to \beta \to \mathit{int}$ erhalten. Für 2 ergibt sich der Typ *int* aus der Regel CON, so dass wir für die Applikation (**fn** $x \Rightarrow$ **fn** $y \Rightarrow x$) 2 mit der Regel APP den Typ $\beta \to \mathit{int}$ erhalten.

Aufgabe 7.2

$$\boldsymbol{T}(\Gamma,(e_1,\ e_2)) = (UT, U\tau_1 \times \tau_2)$$
$$\text{wobei } (T,\tau_1) = \boldsymbol{T}(\Gamma, e_1)$$
$$(U,\tau_2) = \boldsymbol{T}(T\Gamma, e_2)$$

Aufgabe 7.3

([], [], [*LD* 3, *LDC* [*LDV* 1, *LD* 1, *ADD*], *APP*], [])
↦ ([3], [], [*LDC* [*LDV* 1, *LD* 1, *ADD*], *APP*], [])
↦ ([([*LDV* 1, *LD* 1, *ADD*], []), 3], [], [*APP*], [])
↦ ([], [3], [*LDV* 1, *LD* 1, *ADD*], [([], [], [])])
↦ ([3], [3], [*LD* 1, *ADD*], [([], [], [])])
↦ ([1, 3], [3], [*ADD*], [([], [], [])])
↦ ([4], [3], [], [([], [], [])])
↦ ([4], [], [], [])}

Aufgabe 7.4

[*LDC* [*LDC* [*LDV* 1, *LDV* 2, *APP*, *LDV* 2, *APP*]]]

Aufgabe 7.5

```
[ DUM 1,
  LDC
    [LDV 1, LD 3, LT,
     COND
       ([LDV 1, RET],
        [LDV 1, LDV 1, LD 1, SUB, LDV 2, APP, MULT, RET])
    ],
  LDC [LDV 1],
  RAP 1
]
```

Literatur

Appel, A.W. (1992). *Compiling with Continuations*. Cambridge University Press, NewYork, NY.

Backus, J. (1978). Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM 21*, 613-641.

Bird, R.S. (1998). *Introduction to Functional Programming Using Haskell*, 2nd Edition, Prentice-Hall International, London, UK.

Birkedal, L., Rothwell, N., Tofte, M. und Turner, D.N. (1993). The ML Kit (Version 1). Technical Report DIKU 93/14, Department of Computer Science, University of Copenhagen.

Cardelli, L. (1987). Basic Polymorphic Type Checking. *Science of Computer Programming 8*, 147-172.

Damas, L. und Milner, R. (1982). Principal Type Schemes for Functional Programming Languages. *9th ACM Symp. on Principles of Programming Languages*, S. 207-208.

Erwig, M. (1999). *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, München.

Field, A.J. und Harrison, P.G. (1988). *Functional Programming*. Addison-Wesley, Wokingham, UK.

Hall, C.V., Hammond, K. Peyton Jones, S.L., Wadler, P.L. (1996). Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems 18* (2), 109-138.

Henderson, P. (1980). *Functional Programming: Application and Implementation*. Prentice-Hall International, London, UK.

Hudak, P. (1989). Conceptions, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys 21*, 359-411.

Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal 32*, 98-107.

Jones, M.P. (1992). A Theory of Qualified Types. In: Krieg-Brückner, B. (Hrsg.), *4th European Symp. on Programming*, LNCS 582, Springer-Verlag, Berlin, S. 287-306.

Jones, M.P. (1995). A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. *Journal of Functional Programming 5* (1), 1-35.

Landin, P.J. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal 6*, 308-320.

Milner, R. (1978). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences 17*, 248-375.

Milner, R., Tofte, M., Harper, R. und MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA.

Mitchell, J.C. (1990). Type Systems for Programming Languages. In: van Leeuven, J. (Hrsg.), *Handbook of Theoretical Computer Science Vol. B*, Elsevier, Amsterdam, S. 367-458.

Nipkow, T. und Prehofer, C. (1993). Type Checking Type Classes. *20th ACM Symp. on Principles of Programming Languages*, S. 409-418.

Nipkow, T. und Snelting, G. (1991). Type Classes and Overloading Resolution via Order-Sorted Unification. In: Hughes, J. (Hrsg.), *Conf. on Functional Programming and Computer Architecture*, LNCS 523, Springer-Verlag, Berlin, S. 1-14.

Okasaki, C. (1999). *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK.

Peyton Jones, S.L. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall International, Englewood Cliffs, NJ.

Peyton Jones, S.L. und Lester, D.R. (1992). *Implementing Functional Languages: A Tutorial*. Prentice-Hall International, Englewood Cliffs, NJ.

Reade, C. (1989). *Elements of Functional Programming*. Addison-Wesley, Wokingham, UK.

Robinson, J.A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM 12* (1), 23-41.

Shao, Z. (1997a). An Overview of the FLINT/ML Compiler. *ACM SIGPLAN Workshop on Types in Compilation*.

Shao, Z. (1997b). Typed Common Intermediate Format. *USENIX Conference on Domain-Specific Languages*.

Shao, Z. und Appel, A.W. (1995). A Type-based Compiler for Standard ML. *ACM Conf. on Programming Language Design and Implementation*, S. 116-129.

Tarditi, D., Acharya, A. und Lee, P. (1990). No Assembly Required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R. und Lee P. (1996). TIL: A Type-Directed Optimizing Compiler for ML. *ACM Conf. on Programming Language Design and Implementation*, S. 181-192.

Ullman, J.D (1998). *Elements of ML Programming*. Prentice-Hall International, Englewood Cliffs, NJ.

Wadler, P. und Blott, S. (1989). How to Make Ad Hoc Polymorphism Less Ad Hoc. *16th ACM Symp. on Principles of Programming Languages*, S. 60-76.

Wilhelm, R. und Maurer, D. (2007). *Übersetzerbau: Theorie, Konstruktion, Generierung*. 2. Aufl., Springer-Verlag, Berlin.

Index