Übersetzungsprinzipien funktionaler Programmiersprachen

Hochschul-Abschlußarbeit

Technische Universität Chemnitz

Sektion Informatik

Vorgelegt von

Dirk S. Großmann

Geboren am 16. Juli 1966 in Hagenow

Betreuer: *Prof. Dr. sc. nat. Klaus Mätzel*

---

23. April 1991

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der funktionale Programmierstil hat seine Grundlage in den mathematischen Funktionen, den eindeutigen Abbildungen aus einem Definitions- in einen Wertebereich. Für den funktionalen Programmierer ist auch ein Programm nichts weiter als eine Funktion, die die Menge der Eingabedaten in die der Ausgabedaten abbildet. Die Programmabarbeitung besteht in der Anwendung dieser Funktion auf bestimmte Eingaben. Wie das geschieht, interessiert den Programmierer dabei nicht; für ihn ist nur wichtig, daß das Gewünschte passiert.

Dieses Programmiermodell wird von einer Reihe von Programmiersprachen, den funktionalen Sprachen, unterstützt. Ein in einer solchen Sprache geschriebenes Programm enthält neben Beschreibungen der zu verarbeitenden Daten Spezifikationen von Funktionen. Die Funktionsspezifikationen beschreiben Definitions- und Wertebereich, führen formale Parameter als Platzhalter für die Argumente ein und enthalten einen Ausdruck, der die Vorschrift zur Bestimmung des Funktionswertes darstellt. Die Spezifikation einer Funktion ist selbst ein Datenobjekt, ein sogenanntes Funktionsobjekt, das (wie jedes andere Datenobjekt auch) einer Funktion als Argument übergeben, von einer Funktion als Wert geliefert sowie in einer zusammengesetzten Datenstruktur plaziert werden kann.

Ein typisches funktionales Programm besteht aus einer Hierarchie von Funktionen, wobei die niedersten nur vom Programmiersystem bereitgestellte Standard-Funktionen benutzen und höhere auch von bereits definierten Funktionen Gebrauch machen. Die höchste Funktion realisiert mit Hilfe der anderen die gesamte Programmfunktion.

Eine hervorragende Einführung in die funktionale Programmierung gibt Wikström [Wikström 1987], während Henderson [Henderson 1980], Henson [Henson 1987] sowie Field und Harrison [Field 1988] mehr ins Detail gehen. Hughes [Hughes 1989] beleuchtet die Eigenheiten der funktionalen Programmierung näher und beschreibt, warum sie von Bedeutung ist.

Funktionale Programmiersprachen sind eine echte

Alternative [Backus 1978] zu den konventionellen Sprachen, die Abstraktionen der von Neumann-Architektur darstellen: Variablen sind Abstraktionen von Speicherplätzen, Zuweisungen von Lade- und Speicherinstruktionen sowie Fallauswahl-Anweisungen und Schleifen Abstraktionen von Maschinenbefehlsfolgen, Vergleichen und Sprüngen. Bei der konventionellen Programmabarbeitung werden die Anweisungen nacheinander ausgeführt, explizite oder implizite Sprünge bewirken eine Unterbrechung der normalen Abarbeitungsreihenfolge. Dadurch, daß sich Variablenwerte während der Laufzeit des Programms ändern, ist das Ergebnis der Ausführung einer Anweisung von dessen Vorgeschichte abhängig.

Demgegenüber kennen die (reinen) funktionalen Sprachen keine Seiteneffekte, vielmehr entspricht die Programmausführung der Berechnung des Wertes eines Ausdruckes. In jedem Ausdruck kann ein Unterausdruck immer durch einen anderen, der den gleichen Wert liefert, ersetzt werden, ohne daß sich die Bedeutung ändert. Aufgrund des Fehlens von Seiteneffekten hat ein Ausdruck immer denselben Wert, unabhängig von der Auswertungsreihenfolge. Ein Wert kann dabei nicht nur ein elementares Objekt, wie z. B. eine Zahl, sondern auch eine komplexe Datenstruktur sein, unter gewissen Umständen sogar eine potentiell unendliche Liste oder ein potentiell unendlicher Baum.

Unglücklicherweise ist die effiziente Implementation funktionaler Sprachen aufwendiger und schwieriger als die der konventionellen, da bei der Entwicklung der ersteren besonderer Wert auf eine klare Semantik und bequeme Anwendbarkeit gelegt wurde, und nicht auf die Eigenschaften der gegenwärtigen Computer-Hardware. Erst die zukünftigen Multiprozessor-Computer werden funktionale Programme sehr effizient abarbeiten [Peyton Jones 1989], aber bis diese Rechner weit verbreitet sind, dauert es noch längere Zeit. Bis dahin haben wir für unsere funktionalen Programme nur die von Neumann-Rechner.

Diese Literaturstudie beschäftigt sich daher mit der Übersetzung (Compilation) funktionaler Sprachen in von Neumann-Maschinensprachen. Dafür sind zwei Prinzipien bekannt, die umgebungsbasierte Übersetzung mit Hilfe der Closures sowie die Graphenreduktion. Für das Verständnis dieser Arbeit sollte der Leser mit den Grundlagen der funktionalen Programmierung vertraut sein, insbesondere sollten ihm die in der Einleitung gefallenen Begriffe nicht neu vorkommen. Als Einführungslektüre empfehlen wir [Wikström 1987] oder den ersten Teil von [Field 1988].

Das zweite Kapitel geht näher auf die Besonderheiten der funktionalen Sprachen ein und beschreibt insbesondere deren Unterteilung sowie den Lambda-Kalkül.

Das dritte Kapitel beschäftigt sich mit der Transformation von funktionalen Programmiersprachen in Ausdrücke des erweiterten Lambda-Kalküls. Der erweiterte Lambda-Kalkül dient als Zwischensprache auf dem Weg zum ausführbaren Maschinenprogramm.

Im vierten Kapitel wird die umgebungsbasierte Übersetzung von funktionalen Sprachen dargestellt. Der Schwerpunkt liegt hier auf der Behandlung der SECD-Maschine als eine Musterimplementation dafür.

Im fünften Kapitel befassen wir uns mit der Implementation funktionaler Sprachen mit Hilfe der Graphenreduktion. Hier sehen wir uns zuerst die Repräsentation eines funktionalen Programms in Form eines Graphen an, um dann den Prozeß der Reduktion auszuführen. Dazu bedienen wir uns der Technik der Superkombinatoren und reißen kurz die SK-Kombinatoren als alternative Implementationstechnik an.

Das abschließende sechste Kapitel ist der G-Maschine, einer Stack-Maschine für Graphenreduktion, gewidmet. Diese Maschine zeigt, wie Graphenreduktion in eine Form übersetzt werden kann, die für die direkte Ausführung auf einem gewöhnlichen sequentiellen Rechner geeignet ist. Die G-Maschine ist verantwortlich für einen drastischen Geschwindigkeitsgewinn von Implementationen funktionaler Sprachen. Damit kommen die Ausführungszeiten von in funktionalen Sprachen geschriebenen Programmen auf konventionellen Rechnern in Bereiche, die sonst nur den konventionellen Programmen vorbehalten blieben, was vielleicht die Akzeptanz der funktionalen Programmierung unter den Anwendern erhöhen könnte.

Kapitel 2

Besonderheiten der funktionalen Sprachen

In dieser Arbeit wollen wir uns vorwiegend mit den reinen funktionalen Programmiersprachen beschäftigen. Diese enthalten — im Gegensatz zu den funktionalen Sprachen im erweiterten Sinne — weder Seiteneffekte noch andere Kontrollstrukturen außer der Funktionsanwendung. Das zur sogenannten umgebungsbasierten Übersetzung Gesagte kann jedoch sinngemäß auf Sprachen, die Seiteneffekte zulassen, übertragen werden; man muß nur beachten, daß sich der Wert einer Variablen ändern darf. Auf die Behandlung konventioneller Kontrollstrukturen, wie die Schleifen und Sprünge, werden wir an dieser Stelle verzichten; sie erfolgt — falls wirklich nötig — wie bei den konventionellen Sprachen gehabt.

Die funktionalen Sprachen unterscheiden sich nicht prinzipiell voneinander; sie haben ihre theoretische Grundlage im Lambda-Kalkül und sind (mit einigen Bequemlichkeiten für den Programmierer versehene) Auslegungen davon. Wir wenden uns daher zuerst diesem Lambda-Kalkül zu (der Abschnitt darüber ist dem zweiten Kapitel von [Peyton Jones 1987] entnommen; eine ähnliche Beschreibung finden wir auch in [Revesz 1988]).

2.1 Der Lambda-Kalkül

Der von A. Church ursprünglich zur Präzisierung des Berechenbarkeitsbegriffes entwickelte Lambda-Kalkül war die erste „funktionale Programmiersprache“ überhaupt und ist bis heute die Basis für die anderen Sprachen dieser Art. Wir verwenden ihn als Zwischensprache bei der Überführung der „höheren“ funktionalen Sprachen in deren maschinennahe Implementationen. Das hat zwei Gründe.

1. Der Lambda-Kalkül ist eine Sprache mit nur wenigen syntaktischen Konstrukten und mit einfacher Semantik. Das macht ihn geeignet für Betrachtungen über Implementationen, da eine Implementation des Lambda-Kalküls nur wenige Konstrukte zu unterstützen braucht und da es die einfache Semantik erlaubt, die Korrektheit unserer Implementation zu überprüfen.

2. Der Lambda-Kalkül ist ausdrucksstark genug, um alle berechenbaren Funktionen von jedem möglichen Typ und mit beliebig vielen Argumenten zu definieren. Wir können demnach jede funktionale Sprache implementieren, indem wir sie in den Lambda-Kalkül übersetzen.

2.1.1 Syntax des Lambda-Kalküls

In Ausdrücken des Lambda-Kalküls sind alle Funktionsanwendungen in Präfixform geschrieben, z. B. bei

$$+ \quad (* \ 5 \ 6) \quad (* \ 8 \ 3)$$

Vom Standpunkt der Implementation aus stellen wir uns ein funktionales Programm als einen Ausdruck vor, der ausgewertet wird. Die Auswertung erfolgt durch wiederholte Auswahl eines reduzierbaren Ausdrucks und dessen Reduktion. In unserem Beispiel gibt es zwei solcher reduzierbaren Ausdrücke, und zwar $(* \ 5 \ 6)$ und $(* \ 8 \ 3)$. Der Gesamtausdruck ist nicht reduzierbar, da er reduzierbare Ausdrücke, deren Werte für die Reduktion benötigt werden, enthält. Die Reduzierung unseres Beispiels kann wie folgt aussehen.

$$\begin{aligned} + \quad (* \ 5 \ 6) \quad (* \ 8 \ 3) \quad &\longrightarrow \quad + \quad 30 \quad (* \ 8 \ 3) \\ &\longrightarrow \quad + \quad 30 \quad 24 \\ &\longrightarrow \quad 54 \end{aligned}$$

Die Funktionsanwendung wird durch einfache Hintereinanderschreibung ausgedrückt. Dabei reicht es aus, wenn jede Funktion nur ein Argument hat. Im Ausdruck

$$(+ \ 3) \ 4$$

bezeichnet $(+ \ 3)$ eine Funktion, die 3 zu ihrem Argument addiert. Der gesamte Ausdruck ist „die Funktion +, angewendet auf das Argument 3; das Ergebnis davon ist eine Funktion, die auf 4 angewendet wird“. Analog stellen wir uns die Verfahrensweise bei mehreren Argumenten vor. Diese Transformation von Multi-Argument-Funktionen in Folgen von unären Funktionen bezeichnen wir (nach H. Curry) als *Curry*ing.

Wenn eine Funktion zwei oder mehrere Argumente hat, wird eine partielle Anwendung (d. h. eine Anwendung mit zu wenigen Argumenten) als eine Funktion der restlichen Argumente betrachtet.

In seiner reinen Form hat der Lambda-Kalkül keine eingebauten Funktionen und Konstanten, sie werden aber aus praktischen Erwägungen aufgenommen. Zu den eingebauten Funktionen gehören die arithmetischen und logischen Funktionen sowie die Funktion **IF**, die durch folgende Reduktionsregeln definiert ist.

$$\begin{aligned} \textbf{IF} \quad \mathit{TRUE}\ E_1\ E_2 &\longrightarrow E_1 \\ \textbf{IF} \quad \mathit{FALSE}\ E_1\ E_2 &\longrightarrow E_2 \end{aligned}$$

Außerdem nehmen wir Datenkonstruktoren in den Lambda-Kalkül auf, und zwar die Funktionen **CONS**, **HEAD** und **TAIL**. **CONS** konstruiert ein zusammengesetztes Datenobjekt, das mit **HEAD** und **TAIL** auseinandergenommen werden kann.

$$\begin{aligned} \textbf{HEAD} \quad (\textbf{CONS}\ a\ b) &\longrightarrow a \\ \textbf{TAIL} \quad (\textbf{CONS}\ a\ b) &\longrightarrow b \end{aligned}$$

Der Lambda-Kalkül enthält ein Konstrukt, die λ-Abstraktion, um eine neue Funktion zu bezeichnen. Hier ist ein Beispiel dafür.

$$\lambda x\,.\ +\ \ x\ 1$$

Das "λ" wird gefolgt von einer Variablen, dem formalen Parameter, dem nach dem Punkt der Funktionskörper folgt. Durch den obigen Ausdruck wird eine anonyme Funktion von x definiert, die 1 zu ihrem Argument x adiert.

Ein Lambda-Ausdruck ist ein Ausdruck im Lambda-Kalkül und kann die nachstehend angegebenen Formen annehmen.

⟨ausdruck⟩	::=	⟨konst⟩	Konstante
	|	⟨var⟩	Variable
	|	⟨ausdruck⟩ ⟨ausdruck⟩	Funktionsanwendung
	|	λ ⟨var⟩ . ⟨ausdruck⟩	λ-Abstraktion

Im folgenden verwenden wir Kleinbuchstaben für Variablen und Großbuchstaben für ganze Lambda-Ausdrücke sowie für eingebaute Funktionen.

2.1.2 Operationale Semantik

Um mit den Ausdrücken des Lambda-Kalküls zu hantieren, benötigen wir Regeln, die die Überführung von einem Lambda-Ausdruck in einen anderen, der zum ersten äquivalent ist, beschreiben. Doch vorher führen wir die Terminologie der freien und gebundenen Variablen ein.

Zur Auswertung des Ausdrucks

$$(\lambda x\,.\ +\ \ x\ \ y)\quad 4$$

brauchen wir den globalen Wert für y, jedoch keinen globalen Wert für x, da x der formale Parameter der Funktion ist. Wir sagen, daß in unserem Beispiel die Variable x gebunden, die Variable y dagegen frei vorkommt.

Ein Vorkommen einer Variablen in einem Ausdruck ist entweder *frei* oder *gebunden*.

x ist frei in x (aber nicht in jeder anderen Variablen oder Konstanten).

$$
\begin{array}{rcl}
x \text{ ist frei in} (EF) & \Longleftrightarrow & x \text{ ist frei in} E \\
 & & \textit{oder } x \text{ ist frei in } F \\
x \text{ ist frei in} \lambda\, y\,.\, E & \Longleftrightarrow & x \text{ und } y \text{ sind verschiedene Variable} \\
 & & \textit{und } x \text{ ist frei in } E \\
x \text{ ist gebunden in} (EF) & \Longleftrightarrow & x \text{ ist gebunden in} E \\
 & & \textit{oder } x \text{ ist gebunden in } F \\
x \text{ ist gebunden in} \lambda\, y\,.\, E & \Longleftrightarrow & (x \text{ und } y \text{ sind die gleiche Variable} \\
 & & \textit{und } x \text{ ist frei in } E) \\
 & & \textit{oder } x \text{ ist gebunden in } E
\end{array}
$$

Keine Variable kommt in einem Ausdruck, der nur aus einer einzigen Konstanten oder Variablen besteht, gebunden vor.

Allgemein hängt der Wert eines Lambda-Ausdrucks nur von den Werten seiner freien Variablen ab. Ein Vorkommen einer Variablen ist gebunden, wenn es ein äußere Lambda-Abstraktion gibt, die diese Variable bindet; ansonsten ist das Vorkommen frei.

Damit gehen wir zu den Transformationsregeln für Lambda-Ausdrücke über. Die erste ist die *Alpha-Transformation*, die es uns erlaubt, den Namen des formalen Parameters einer jeden Lambda-Abstraktion gegen einen neuen zu ersetzen, wenn dieser Name nicht frei im Körper vorkommt und wenn wir das korrekt tun. Formal läßt sich das wie folgt beschreiben.

$$(\lambda\, x\,.\, E) \quad \longleftrightarrow_\alpha \quad (\lambda\, y\,.\, E[y/x])$$

Die Schreibweise $E[M/x]$ bedeutet den Ausdruck E, in dem alle freien Vorkommen von x durch M ersetzt sind. Die genaue Definition folgt unten.

Die *Beta-Transformation* gibt an, wie eine Lambda-Abstraktion auf ihr Argument angewendet wird.

$$(\lambda\, x\,.\, E)\ M \quad \longleftrightarrow_\beta \quad E[M/x]$$

Die *Eta-Transformation* verwendet man zur Vereinfachung von Lambda-Ausdrücken. Wenn x nicht frei in E ist und E eine Funktion bezeichnet, dann ist

$$(\lambda\, x\,.\, E\ x) \quad \longleftrightarrow_\eta \quad E$$

Zuletzt geben wir noch die Definition der *Substitution* an.

$$
\begin{array}{rcl}
x[M/x] & = & M \\
c[M/x] & & \text{wobei c ungleich x ist} \\
 & = & c \\
(EF)[M/x] & = & E[M/x]F[M/x] \\
(\lambda x.E)[M/x] & = & \lambda x.E \\
(\lambda y.E)[M/x] & & \text{wobei y eine Variable ungleich x ist} \\
 & = & \lambda y.E[M/x] \\
 & & \text{wenn x nicht frei in E oder y nicht} \\
 & & \text{frei in M ist} \\
 & = & \lambda z.(E[z/y])[M/x] \qquad \text{sonst} \\
 & & \text{wobei z eine neue Variable ist, die nicht} \\
 & & \text{frei in E oder M vorkommt}
\end{array}
$$

2.1.3 Bezeichnende Semantik

Es gibt zwei verschiedene Sichten auf eine Funktion: als einen Algorithmus, der aus einem gegebenen Argument einen Wert produziert, oder als eine Menge von geordneten Argument-Wert-Paaren. Die erste Sicht ist dynamisch oder operational, in ihr erscheint eine Funktion als eine Folge von Operationen. In der zweiten, der statischen oder bezeichnenden Sicht, wird eine Funktion als eine feste Menge von Beziehungen zwischen Argumenten und den entsprechenden Werten aufgefaßt.

Der Zweck der bezeichnenden Semantik (*denotational semantics*) einer Sprache ist es, jedem Ausdruck in dieser Sprache einen Wert zuzuweisen. Wir können daher die Semantik einer Sprache als eine mathematische Funktion **Eval** von der Menge der Ausdrücke in die Menge der Werte angeben. Die bezeichnende Semantik des Lambda-Kalküls ist wie folgt definiert.

$$
\begin{array}{rcl}
\mathbf{Eval}[k]\ p & = & k \\
\mathbf{Eval}[f]\ p & = & \text{Entsprechend der Semantik der} \\
 & & \text{eingebauten Funktion } f \\
\mathbf{Eval}[x]\ p & = & p(x) \\
\mathbf{Eval}[E_1\ E_2]\ p & = & (\mathbf{Eval}[E_1]\ p)\quad(\mathbf{Eval}[E_2]\ p) \\
\mathbf{Eval}[\lambda\, x\, .\, E]\ p\ a & = & \mathbf{Eval}[E]\quad p\,[x \leftarrow a]
\end{array}
$$

wobei k eine Konstante, f eine eingebaute Funktion und x eine Variable ist, E, E_1 und E_2 Ausdrücke sind sowie p die Umgebung (Kontextinformation), d. h. eine Funktion, die Variablennamen auf ihre Werte abbildet, ist. Die Schreibweise $p\,[x \leftarrow a]$ bedeutet, daß die Variable x mit dem Wert a aktualisiert oder, falls x noch nicht existierte, das Paar (x, a) zu p hinzugefügt wurde.

2.2 Der erweiterte Lambda-Kalkül

Der erweiterte Lambda-Kalkül ist eine Obermenge des gewöhnlichen Lambda-Kalküls, so daß jeder Ausdruck im Lambda-Kalkül auch einer im erweiterten Lambda-Kalkül ist. Zusätzlich enthält der erweiterte Lambda-Kalkül mustererkennende Lambda-Abstraktionen, den Infix-Operator „fetter Balken“ (FATBAR II), **case**-Ausdrücke sowie Ausdrücke, um Lambda-Abstraktionen zu benennen und wechselseitig rekursive Funktionen einfach zu definieren. Wir wollen uns die hinzugekommenen Ausdrücke der Reihe nach ansehen.

2.2.1 Ausdrücke zur Behandlung der Mustererkennung

Die ersten drei neuen Konstrukte dienen der Unterstützung der Mustererkennung. Obwohl wir uns in dieser Arbeit nicht mit der Mustererkennung befassen wollen, geben wir der Vollständigkeit halber einen Überblick über strukturierte Typen und Muster.

Ein *Typ* ist die Bezeichnung für eine Menge von Datenobjekten; „strukturiert“ bedeutet, daß die Elemente des Typs zusammengesetzt sind — etwa im Sinne der *Records* und *Varianten-Records* von *Pascal*. Zu jedem strukturierten Typ gibt es *Konstruktoren*, die verwendet werden, um die Datenobjekte des Typs zu konstruieren.

Es gibt *Produkttyp-* und *Summentyp-Konstruktoren*. Einen Produkttyp kann man sich als Äquivalent eines *Record*-Typs in *Pascal* vorstellen. Er hat den Aufbau

$$Tp \quad = \quad c \quad T_1 \quad \cdots \quad T_m$$

wobei c der Typkonstruktor ist und die T_i Typen sind. Ein Summentyp ist eine Art *Varianten-Record* und wie folgt aufgebaut.

$$Ts \quad = \quad T_1 \quad | \quad T_2 \quad | \quad \cdots \quad | \quad T_n$$

Ein Objekt des Typs Ts kann demnach vom Typ T_1 oder vom Typ T_2 oder ... oder vom Typ T_n sein. Allgemein sieht die Definition eines strukturierten Typs dann so aus.

$$\begin{array}{rcl} T & = & c_1 \quad T_{1,1} \quad \cdots \quad T_{1,r_1} \\ & | & \cdots \\ & | & c_n \quad T_{n,1} \quad \cdots \quad T_{n,r_n} \end{array}$$

wobei die c_i Konstruktoren der Stelligkeit r_i und die $T_{i,j}$ Typen sind.

Mit Hilfe der Typen können wir nun die Muster definieren, da wir die Muster verwenden wollen, um Teilmengen der Objekte eines Typs zu bilden. Wir sagen, daß ein Datenobjekt zu der durch das Muster spezifizierten Teilmenge gehört, wenn das Muster auf das Datenobjekt paßt.

Ein *Muster* ist entweder eine Konstante, eine Variable oder ein Konstruktor-Muster der Form $(c\ p_1\ \ldots\ p_r)$, wobei c ein Konstruktor der Stelligkeit r ist und die p_i ihrerseits wieder Muster sind.

Hat ein Muster die Form $(s\ p_1\ \ldots\ p_r)$, wobei s ein Summentyp-Konstruktor ist, nennen wir es *Summenkonstruktor-Muster*. Ein Muster der Form $(t\ p_1\ \ldots\ p_r)$, wobei t ein Produkttyp-Konstruktor ist, heißt *Produktkonstruktor-Muster*.

Bei einer mustererkennenden Lambda-Abstraktion tritt als formaler Parameter ein Muster auf, es ergibt sich die Form

$$\lambda\, p\,.\, E$$

Wir geben nun die Semantik der mustererkennenden Lambda-Abstraktion an, und zwar für jede Form des Musters p.

Ist das Muster eine Variable, haben wir eine gewöhnliche Lambda-Abstraktion vor uns.

Wenn das Muster eine Konstante k ist, gibt es drei Möglichkeiten.

$$\begin{array}{lcll}
\mathbf{Eval}[\lambda\, k\,.\, E]\ a & = & \mathbf{Eval}[E] & \text{wenn} \quad a = \mathbf{Eval}[k] \\
\mathbf{Eval}[\lambda\, k\,.\, E]\ a & = & \text{FAIL} & \text{wenn} \quad a \neq \mathbf{Eval}[k] \quad \text{und} \quad a \neq \bot \\
\mathbf{Eval}[\lambda\, k\,.\, E]\ \bot & = & \bot &
\end{array}$$

Das Symbol "$\bot$" bedeutet, daß die Auswertung nicht terminiert; der spezielle Wert FAIL signalisiert, daß das Muster nicht auf das Argument paßt.

Als nächstes betrachten wir den Fall des Summenkonstruktor-Musters.

$$\begin{array}{rcll}
\mathbf{Eval}[\lambda(s\ p_1\ \ldots\ p_r).E]\ \ (s\ a_1 \ldots a_r) & = & \mathbf{Eval}[\lambda\, p_1 \ldots \lambda\, p_r.E]\ a_1\ \ldots\ a_r & \\
\mathbf{Eval}[\lambda(s\ p_1\ \ldots\ p_r).E]\ \ (s'\ a_1 \ldots a_r) & = & \text{FAIL} & \text{wenn } s \neq s' \\
\mathbf{Eval}[\lambda(s\ p_1\ \ldots\ p_r).E]\ \ \bot & = & \bot &
\end{array}$$

Zuletzt geben wir die Semantik-Regel für das Produktkonstruktor-Muster an.

$$\begin{array}{rcl}
\mathbf{Eval}[\lambda(t\ p_1\ \ldots\ p_r).E]\ \ a & = & \mathbf{Eval}[\lambda\, p_1 \ldots \lambda\, p_r.E] \\
 & & (\text{SEL-t-1}\ a) \\
 & & \ldots \\
 & & (\text{SEL-t-R}\ a)
\end{array}$$

Hier ist SEL-t-i eine eingebaute Funktion, die das i-te Feld von einem strukturierten Objekt, das mit dem Konstruktor t erzeugt wurde, auswählt.

Die mustererkennenden Lambda-Abstraktionen möchten wir in einer neuen Funktion in einer solchen Art verknüpfen, daß sie versucht, der Reihe nach verschiedene mustererkennende Lambda-Abstraktionen auf das Argument anzuwenden. Als Wert liefert sie das Resultat der Anwendung der ersten mustererkennenden Lambda-Abstraktion, auf die das Argument paßt. Hier hilft uns der II-Operator, der zu folgendem Ergebnis führt.

$$\begin{array}{lcll} f & = & \lambda\, x\,. & (\quad (\lambda\, p_1\,.\,E_1)\ x \\ & & & \text{II}\quad (\lambda\, p_2\,.\,E_2)\ x \\ & & & \text{II}\quad \ldots \\ & & & \text{II}\quad (\lambda\, p_n\,.\,E_n)\ x \\ & & & \text{II}\quad \textbf{ERROR}) \end{array}$$

Versagen alle $(\lambda\, p_i\,.\,E_i)$, wird ERROR als Wert geliefert.

Die Funktion II ist eine Infix-Funktion, deren Verhalten durch folgende Semantik-Gleichungen beschrieben ist.

$$\begin{array}{rcl} a \ \text{II}\ b & = & a \qquad \text{wenn} \quad a \neq \bot \quad \text{und} \quad a \neq \text{FAIL} \\ \text{FAIL}\ \text{II}\ b & = & b \\ \bot\ \text{II}\ b & = & \bot \end{array}$$

Eine Form der Beschreibung einfacher Mustererkennung sind die **case**-Ausdrücke, die bei der Verarbeitung von Objekten von Summentypen verwendet werden, um die Teiltypen getrennt zu betrachten. Der allgemeine Aufbau des **case**-Ausdrucks ist

$$\begin{array}{lll} \textbf{case} & E & \textbf{of} \\ & c_1 \quad v_{1,1} \ \ldots\ v_{1,r_1} & \Longrightarrow \quad E_1 \quad ; \\ & \ldots & \\ & c_n \quad v_{n,1} \ \ldots\ v_{n,r_n} & \Longrightarrow \quad E_n \\ \textbf{end} & & \end{array}$$

wobei v eine Variable ist, die E_i Ausdrücke und die $v_{i,j}$ unterschiedliche Variablen sind und die c_i eine komplette Familie von Konstruktoren einer strukturierten Typdefinition darstellen. Die Muster dürfen also nicht geschachtelt und müssen erschöpfend sein.

2.2.2 Let- und letrec-Ausdrücke

Die **let**- und **letrec**-Ausdrücke dienen dazu, ansonsten anonyme Lambda-Ausdrücke zu benennen. Wir beginnen mit den **let**'s, die die folgende Syntax haben.

$$\textbf{let} \quad v \ = \ B \quad \textbf{in} \quad E$$

wobei v eine Variable ist und B sowie E Ausdrücke in erweiterter λ-Notation sind. Der **let**-Ausdruck führt eine Definition für eine Variable v ein, die v an B in E bindet. Die Definition $v = B$ ist in E, nicht aber in B sichtbar.

Die Syntax des **letrec**-Ausdrucks ist ähnlich der für **let**.

letrec
$$\begin{array}{lcl} v_1 & = & E_1 \\ \ldots & & \\ v_n & = & E_n \end{array}$$
in E

Das „*letrec*“ steht für „*rekursives let*“ und führt möglicherweise rekursive Bindungen für eine Menge von Variablen v_i ein. Die v_i sind demnach sowohl in E als auch in den E_i sichtbar.

In den **let(rec)**'s dürfen die v_i auch Muster sein, aber damit wollen wir uns hier nicht weiter beschäftigen.

2.3 Getypte und ungetypte Sprachen

Der erweiterte Lambda-Kalkül ist eine ungetypte Sprache, da er seinen Ausdrücken keine Typzwänge auferlegt. Die modernen funktionalen Sprachen sind dagegen streng getypt. Eine Sprache ist streng getypt, wenn ihr Compiler garantieren kann, daß die von ihm akzeptierten Programme ohne Typfehler ausgeführt werden. Trotzdem braucht der Programmierer meist nicht die Typen der von ihm definierten Objekte anzugeben. Der Compiler kann diese Arbeit erledigen, sofern das gesamte Programm konsistent getypt werden kann. Der Teil des Compilers, der das tut, wird als Typchecker bezeichnet. Er versucht, die Typen von Ausdrücken im Programm aus dem Kontext abzuleiten.

Polymorphe Sprachen gestatten die Vereinbarung von polymorphen Funktionen. Der Typ einer solchen Funktion enthält Variable, so daß diese Funktion auf Argumente unterschiedlichen Typs angewendet werden kann.

Ausgezeichnete Diskussionen von Typen und der verschiedenen Arten von Polymorphismus finden wir in [Cardelli 1985] sowie in [Reynolds 1985].

2.4 Strikte und verzögerte Auswertung

Bei der Beschreibung der Semantik des Lambda-Kalküls haben wir nicht festgelegt, wann bei der Funktionsanwendung $(f\,x)$ das Argument ausgewertet wird. Für den Auswertungszeitpunkt gibt es zwei Möglichkeiten, die eine Unterteilung der funktionalen Programmiersprachen bedingen.

Sprachen mit strikter Auswertung werten das Argument zuerst vollständig aus und wenden dann die Funktion auf den Wert des Arguments an.

Bei den Sprachen mit verzögerter Auswertung werden Argumente von Funktionen erst dann ausgewertet, wenn deren Wert unbedingt benötigt wird, und

nicht, bevor die Funktion angewendet wird. Das gilt auch für die Datenkonstruktoren (**CONS**), wobei die Komponenten erst ausgewertet werden, wenn sie extrahiert und benutzt werden, und nicht bei der Erzeugung des Datenobjektes. Aus Effizienzgründen sollte die Auswertung eines Arguments höchstens einmal erfolgen, weitere Zugriffe auf das Argument innerhalb der Funktion benutzen den Wert, der bei der ersten Auswertung berechnet wurde. Bei einer reinen funktionalen Sprache kann man davon ausgehen, daß dieses Verfahren denselben Wert wie eine erneute Auswertung liefert.

Die verzögerte Auswertung bedingt die Notwendigkeit, unausgewertete Ausdrücke zu repräsentieren. Solche Repräsentationen werden auch bei den strikten Sprachen gebraucht, da bei der Auswertung einer Lambda-Abstraktion ein Funktionsobjekt entsteht, dessen Körper ein unausgewerteter Ausdruck ist. Das gleiche Problem tritt auch bei der **IF**-Funktion auf, da wir nicht wollen, daß beide Zweige vor der Funktionsanwendung ausgewertet werden.

Kapitel 3

Übersetzung in den Lambda-Kalkül

Wir haben bereits im letzten Kapitel gesehen, daß der Lambda-Kalkül zur Implementation der funktionalen Sprachen geeignet ist, und wollen ihn daher als Zwischensprache verwenden. Für die Übersetzung gibt es zwei Wege.

1. Wir führen den größten Teil der Übersetzung in der Quellsprache aus, d. h. als Ergebnis jeder einzelnen Transformation entsteht ein einfacheres Programm in der Quellsprache. Das resultierende Programm wird dann in einem letzten Schritt durch nicht viel mehr als eine Syntaxänderung in den Lambda-Kalkül überführt.

2. Wir transformieren zuerst das Quellprogramm durch Syntaxänderung in den erweiterten Lambda-Kalkül. Die Hauptarbeit erledigen wir dann durch wiederholte Transformation zu immer einfacheren Formen, bis wir zu gewöhnlichen Lambda-Ausdrücken kommen.

Obwohl wir bei der Verwendung des ersten Weges keine zusätzliche Sprache definieren müssen, verwenden wir den zweiten; und zwar aus den folgenden Gründen.

1. Funktionale Sprachen sind Sprachen für Programmierer, nicht für Compiler; einige lassen daher gewisse Merkmale vermissen, die für die transformationsbasierte Compilierung wünschenswert sind (z. B gibt es in *Miranda*[1] keine explizite Lambda-Abstraktion).

2. Zu einem höheren Grade als bei den imperativen Sprachen sind die funktionalen Sprachen syntaktische Versionen voneinander, mit relativ weni-

[1]Das *Miranda*-System ist ein eingetragenes Warenzeichen der Firma Research Software Ltd.

gen semantischen Unterschieden. Bei der Verwendung der zweiten Methode können die meisten Transformationen für alle funktionalen Sprachen verwendet werden, und nur der (einfachere) Teil der Übersetzung in den erweiterten Lambda-Kalkül ist für jede Sprache spezifisch. Da wir in dieser Arbeit möglichst allgemein bleiben wollen, bietet sich die zweite Methode geradezu an.

3.1 Übersetzung in den erweiterten Lambda-Kalkül

Die Übersetzung der Quellsprache in den erweiterten Lambda-Kalkül beinhaltet neben der Syntaxänderung die Behandlung der Mustererkennung und der ZF-Ausdrücke, sofern sie von der Quellsprache unterstützt werden.

In den modernen funktionalen Programmiersprachen wie *Hope* [Burstall 1980], *Standard-ML* [Wikström 1987], *Lazy-ML* [Augustsson 1987] oder *Miranda* [Turner 1985, Turner 1987], wird die Mustererkennung verwendet, um die Argumente von Funktionen zu spezifizieren bzw. um zusammengesetzte Datenobjekte zu zerlegen. Ein Beispiel dafür ist die folgende, in *Miranda* geschriebene, Funktion, die die Länge einer Liste berechnet.

```
length []      =  0
length (x:xs)  =  1 + (length xs)
```

Die Mustererkennung läßt sich in geschachtelte **case**-Ausdrücke überführen [Peyton Jones 1987, Augustsson 1987].

Ein weiteres Merkmal verschiedener funktionaler Sprachen ist das Listenverständnis (*list comprehensions*). Listenverständnisse werden auch, als Analogie zum Mengenverständnis der Zermelo-Frankel-Mengentheorie, als ZF-Ausdrücke bezeichnet. Der folgende Ausdruck ist ein Beispiel dafür in *Miranda*

```
[square x | x <- xs; odd x]
```

der die Liste der Quadrate aller ungeraden Zahlen x aus der Menge xs darstellt. Auch die ZF-Ausdrücke müssen an dieser Stelle behandelt werden [Peyton Jones 1987, Augustsson 1987], da der erweiterte Lambda-Kalkül dafür keine Konstrukte vorsieht.

Die Übersetzungsphase bis zum erweiterten Lambda-Kalkül wird zweckmäßig in zwei Schritte aufgeteilt. Zuerst erstellen wir aus dem Eingabestrom eine interne Repräsentation des Quellprogramms. Das geschieht, wie auch bei den konventionellen Compilern, mit Hilfe eines Lexik- und eines Syntaxanalysators, die natürlich mit entsprechenden Werkzeugen wie **Lex** und **Yacc** generierbar sind [Augustsson 1987]. Danach beseitigen wir in dieser Repräsentation die komplexen Muster sowie die ZF-Ausdrücke und überführen

das Ergebnis mit Hilfe geeigneter Transformationsregeln [Peyton Jones 1987] in den erweiterten Lambda-Kalkül.

Indem wir die Informationen über Datentypen speichern, legen wir die Grundlage für die Typprüfung, die jedoch über Ausdrücken im erweiterten Lambda-Kalkül vorgenommen werden kann. Ein Algorithmus zur Tyypprüfung ist in [Milner 1978] und, darauf aufbauend, in [Peyton Jones 1987] beschrieben.

3.2 Übersetzung des erweiterten Lambda-Kalküls

Da wir uns nicht mit der Mustererkennung beschäftigen wollen, betrachten wir hier nur die **let(rec)**-Ausdrücke (dieser Abschnitt geht auf [Landin 1964] und [Peyton Jones 1987] zurück). Für die **let**'s ist die Transformation ziemlich einfach, denn der Ausdruck

$$\textbf{let} \quad v \;=\; B \quad \textbf{in} \quad E$$

ist äquivalent zum Ausdruck

$$(\lambda\, v\,.\, E) \quad B$$

Schwieriger wird es bei den **letrec**'s, da wir hier mehrere Definitionen sowie Rekursion zulassen wollen. Mehrere Vereinbarungen beseitigen wir wie folgt.

$$\begin{array}{rrcl} \textbf{letrec} & v_1 & = & B_1 \\ & & \ldots & \\ & v_n & = & B_n \\ \textbf{in} & E & & \end{array}$$

überführen wir in

$$\textbf{letrec} \quad (t\ v_1\ \ldots\ v_n) \quad = \quad (t\ B_1\ \ldots\ B_n) \quad \textbf{in} \quad E$$

wobei t ein Produkttyp-Konstruktor der Stelligkeit n ist. Wir erhalten einen **letrec**-Ausdruck mit nur einer Definition, der aber jetzt ein Muster auf der linken Seite der Gleichung enthält.

Es bleibt, die Rekursion zu beseitigen. Dazu betrachten wir den Ausdruck

letrec

FAC = λ n . IF (= n 0) 1 (* n (FAC (- n 1)))

in ...

Wir konzentrieren uns hier nur auf die rekursive Definition

$$\text{FAC} \quad = \quad \lambda\, n\,.\,(\cdots \text{FAC} \cdots)$$

Nach der Ausführung einer Beta-Abstraktion erhalten wir

$$\text{FAC} \quad = \quad (\lambda\, \text{fac}\, \lambda\, n\,.\,(\cdots \text{fac} \cdots)) \quad \text{FAC}$$

Wenn wir den Lambda-Ausdruck $(\lambda\, \text{fac}\,.\,\ldots)$ mit F bezeichnen, kommen wir zu der Form

$$\text{FAC} \quad = \quad F \quad \text{FAC}$$

Diese zirkulare Definition besagt, daß ein Objekt (FAC) unverändert bleibt, wenn es von der Funktion F transformiert wird. FAC ist also ein *Fixpunkt* von F. Wir verwenden nun den Buchstaben $\mathbf{Y}$, um die Funktion zu bezeichnen, die den Fixpunkt einer gegebenen Funktion findet. $\mathbf{Y}$ hat das Verhalten, daß

$$\mathbf{Y}\ F \quad = \quad F\ (\mathbf{Y}\ F)$$

ist, und wird als *Fixpunkt-Kombinator* bezeichnet. Mit Hilfe von $\mathbf{Y}$ kann unsere zirkulare Definition so geschrieben werden, daß sie formal nicht länger zirkular ist.

$$\text{FAC} \quad = \quad \mathbf{Y}\ F$$

Wenn wir F wieder ersetzen, bekommen wir

$$\text{FAC} \quad = \quad \mathbf{Y}(\lambda\, \text{fac}\,.\,\lambda\, n\,.\,(\cdots \text{fac} \cdots))$$

bzw. da diese Definition formal nicht mehr zirkular ist, den **let**-Ausdruck

let

$$\text{FAC} \quad = \quad \mathbf{Y} \quad (\lambda\, \text{fac}\,.\ \lambda\, \text{n}\,.\,\text{IF}\ (=\ \text{n}\ 0)\ 1\ (*\ \text{n}\ (\text{fac}\ (\text{-}\ \text{n}\ 1))))$$

in $\ldots$

Das erhaltene Ergebnis verallgemeinern wir nun auf das Problem der Überführung von

$$\textbf{letrec} \quad p \ = \ B \quad \textbf{in} \quad E$$

wobei p ein Muster ist, und bekommen als äquivaltentes **let** den Ausdruck

$$\textbf{let} \quad p \ = \ \mathbf{Y}\ (\lambda\, p\,.\,B) \quad \textbf{in} \quad E$$

Wir verwenden zur Bezeichnung des formalen Parameters in der Lambda-Abstraktion denselben Namen wie auf der linken Seite der Gleichung, damit wir den Ausdruck B, in dem bestimmt die Komponenten des durch diesen Namen bezeichneten Musters vorkommen bzw. dieser Name selbst, unverändert lassen können.

Interessant ist der Fakt, daß der **Y**-Kombinator als λ-Abstraktion definiert werden kann, ohne Rekursion zu benutzen.

$$\mathbf{Y} \quad = \quad \lambda f\,.(\lambda x\,.\,f\ \ (x\ \ x))\quad (\lambda x\,.\,f\ \ (x\ \ x))$$

Durch Anwenden dieser Definition auf ein Argument kann sich der Leser davon überzeugen, daß unser **Y** das geforderte Verhalten besitzt.

Kapitel 4

Umgebungsbasierte Übersetzung

Das umgebungsbasierte Übersetzungsschema ist abgeleitet aus den Erfahrungen bei der Implementation von *Lisp*; es ist das ältere der beiden Übersetzungsprinzipien. Es eignet sich besonders für strikte Sprachen, kann aber auch, wie wir noch sehen werden, für die verzögerte Auswertung verwendet werden. Dieses Schema liegt im Umgebungsbegriff und den Closures als Repräsentation von Funktionsobjekten begründet.

4.1 Umgebungen und Closures

Wir setzen hier und im folgenden voraus, daß das zu übersetzende Programm im erweiterten Lambda-Kalkül vorliegt.

Bei einer Funktionsanwendung im Lambda-Kalkül werden alle freien Vorkommen des formalen Parameters im Funktionskörper gegen das entsprechende Argument ersetzt (*literale Substitution*). Das geschieht durch Instanziierung des Körpers: Es wird eine Kopie (Instanz) des Körpers erzeugt, in die anstelle der formalen Parameter die Argumente eingetragen sind. Die Instanz wird sodann reduziert und stellt nach der Reduktion den Wert der Funktionsanwendung dar. Diese Vorgehensweise ist ziemlich zeitaufwendig, besonders, wenn die Funktion freie Variable enthält.

Wir gehen deshalb einen anderen Weg. Bei der Definition einer Funktion bilden wir eine Menge, die alle aktuellen Bindungen aller in der entsprechenden Lambda-Abstraktion frei vorkommenden Variablen enthält. Eine Bindung ist ein geordnetes Paar, bestehend aus dem Variablennamen und dem (aktuellen) Wert der Variablen; eine Menge von Bindungen eine Umgebung. Das Objekt, das bei der Zusammenfassung der Repräsentation der Lambda-Abstraktion mit der Menge der freien Bindungen entsteht, ist eine *Closure* (Abschluß).

Bei der Anwendung einer Closure auf ihr Argument (verallgemeinert ihre Argumente) passiert folgendes. Zuerst wird die alte Umgebung gegen die in der Closure gespeicherten ersetzt und diese durch die Bindungen der formalen Parameter an die entsprechenden Argumente erweitert. In dieser neuen Umgebung wird der Körper der Lambda-Abstraktion ausgewertet, wobei der Wert berechnet wird. Danach ist die alte Umgebung wieder aktuell.

Diese Verfahrensweise bedingt, daß ein Funktionsobjekt immer in der Umgebung, die bei dessen Definition aktuell war (natürlich erweitert durch die Bindungen der formalen Parameter), ausgewertet wird (statische Variablenbindung).

4.2 Verfahrensweise bei der Übersetzung

Ein funktionales Programm besteht aus einer Menge von globalen Definitionen, die zusammen mit den eingebauten die globale Umgebung bilden, und einem auszuwertenden Ausdruck. Interaktive Programmiersysteme arbeiten meist inkrementell: Jede neue globale Definition (Bindung) erweitert die globale Umgebung, in der Ausdrücke ausgewertet werden.

Da wir die Rekursion zulassen wollen (und müssen), kann man ein funktionales Programm vereinfacht durch den folgenden erweiterten Lambda-Ausdruck darstellen.

let $v_1 = E_1$

in let $v_2 = E_2$

in ...

in letrec

$v_{k+1} = E_{k+1}$

...

$v_n = E_n$

in E

Unser umgebungsbasierter Compiler muß also den Code für den Aufbau der Datenstruktur der globalen Umgebung aufbauen und für die Berechnung des Wertes des Ausdrucks E generieren. Beides bildet dann das übersetzte Programm.

4.2.1 Aufbau der globalen Umgebung

Wir betrachten der Einfachheit halber zuerst die nicht rekursiven Definitionen, werten also die geschachtelten **let**'s nacheinander aus. Die Schachtelung besagt dabei, daß alle früheren Definitionen verfügbar sind.

Als rechte Seite der Vereinbarungen können Lambda-Abstraktionen oder Ausdrücke, die keine Funktionen bezeichnen, stehen. Lambda-Abstraktionen werten wir folgendermaßen zu Closures aus. Wir setzen hier voraus, daß unser Compiler den durch die Lambda-Abstraktion eingekapselten Funktionskörper unter Berücksichtigung der formalen Parameter in ein Stück Maschinencode übersetzt hat. Dieses wird dann mit der aktuellen Umgebung zu einer Closure verbunden.

Stehen als rechte Seiten Ausdrücke, die zu anderen Datenobjekten ausgewertet werden, fügt der Compiler den entsprechenden Code ein. Das ist möglich, da wir es an dieser Stelle mit strikter Auswertung zu tun haben. Der Code bestimmt dann während der Laufzeit das entsprechende Datenobjekt.

Aus dem Variablennamen und der Repräsentation des mit ihm assoziierten Wertes bilden wir ein Paar, das wir als zusätzliches Element in die Umgebung aufnehmen, die dadurch erweitert wird. Aus dieser Vorgehensweise ergibt sich, eine Umgebung als Assoziationsliste darzustellen.

Wir kommen nun zur Behandlung des innersten **letrec**-Ausdrucks, die etwas komplizierter ist, da die rechten Seiten der Definitionen in einer Umgebung ausgewertet werden müssen, die alle zu definierenden Variablen enthält. Wir wollen hier nicht den **Y**-Kombinator verwenden, sondern wählen eine effizientere Implementation. Zuerst erweitern wir die aktuelle Umgebung um so viele Paare, wie es Definitionen gibt. Die Namenskomponente des Paares enthält den entsprechenden Variablennamen, die Wertkomponente ist noch nicht belegt. In dieser Pseudoumgebung werten wir die rechten Seiten nacheinander aus. Bei deren Auswertung dürfen nur Closures, die als Umgebung unsere Pseudoumgebung enthalten, entstehen, da wir jetzt noch nicht auf die Werte der zu definierenden Variablen zugreifen können. Diese Closures werden dann in die entsprechenden Wertkomponenten eingetragen, so daß nach der Auswertung der letzten Definition aus unserer Pseudoumgebung eine richtige Umgebung geworden ist. Stehen in den gebildeten Closures nur Zeiger auf die Umgebung, sind wir an dieser Stelle fertig. Bei Sprachen, die Neubindungen zulassen, steht dagegen in jeder neuen Closure noch die Pseudoumgebung, in die aber jetzt die neuen Werte eingetragen werden können. Damit sind die neuen Closures eventuell zyklisch: Sie enthalten in der Umgebungskomponente eine Variable, deren Wert die Closure selbst ist.

Das hier beschriebene Verfahren können wir selbstverständlich auch für die Auswertung innerer **let(rec)**-Ausdrücke verwenden.

4.2.2 Übersetzung eines Ausdrucks

Ein Ausdruck kann neben den bereits betrachteten **let(rec)**'s und Abstraktionen eine Konstante, eine Variable oder eine Funktionsanwendung sein. Im Falle einer Konstanten ist der Wert deren Repräsentation oder ein Zeiger darauf, bei einer Variablen sehen wir in der Umgebung nach, um den Wert zu erhalten.

Bei der Übersetzung einer Funktionsanwendung unterscheiden wir die Überführung in einen Unterprogrammruf und die Behandlung gewisser Standard-Funktionen. Bei beiden erzeugen wir zuerst den Code zur Berechnung der Argumentwerte.

Bevor wir im ersten Falle zum Funktionscode verzweigen, retten wir die aktuelle Umgebung, zum Beispiel auf einen Stack, und erklären die in der Closure gespeicherte Umgebung zur aktuellen. Diese erweitern wir durch die Bindungen der formalen Parameter. Werte von formalen Parametern, die nur lokal in der Funktion verwendet werden (also nicht über Closures nach außen gegeben werden), können wir auch, wie bei konventionellen Sprachen üblich, über einen Stack übergeben und im Funktionskörper diese formalen Parameter durch Bezüge auf den Stack ersetzen. Dadurch lassen sich eventuell einige Umgebungswechsel ganz einsparen.

Nach der Parameterübergabe springen wir den Funktionscode an, der den Wert berechnet. Diesen geben wir z. B. über den Stack oder ein Register zurück und stellen die alte Umgebung wieder her.

Bestimmte Standard-Funktionen können wir auch direkt codieren, d. h. entsprechenden Code in den laufenden Code einfügen. Deren Parameter machen meist keine Schwierigkeiten, so daß sie im Stack oder sogar in Registern gehalten werden können, oder erfordern eine Sonderbehandlung.

4.3 Die SECD-Maschine

Die SECD-Maschine, die von P. Landin [Landin 1964] erfunden wurde, ist eine abstrakte Maschine für die Ausführung von reinen funktionalen Sprachen. Sie ist eine Musterimplementation des umgebungsbasierten Schemas und für die Verarbeitung von strikten Sprachen vorgesehen, kann aber, wie wir im nächsten Abschnitt sehen werden, dahingehend erweitert werden, daß sie auch verzögerte Auswertung zuläßt.

Die beiden Abschnitte über die SECD-Maschine haben wir [Henderson 1980] entnommen; dort findet der Leser auch Beispiele über die Wirkungsweise dieser Maschine.

Ein Programm für die SECD-Maschine ist ein S-Ausdruck, bestehend aus Maschineninstruktionen und Operanden, der eine Funktion beschreibt. Mit dem Programm berechnet die Maschine aus der S-Ausdrucks-Repräsentation der Funktionsargumente die S-Ausdrucks-Repräsentation des Ergebnisses der Anwendung der Funktion auf die Argumente.

Der Begriff des S-Ausdrucks ist der Sprache *Lisp* entlehnt; ein S-Ausdruck ist entweder ein atomares Objekt (eine Konstante bzw. Variable) oder ein gepunktetes Paar, dessen Komponenten S-Ausdrücke sind.

Die SECD-Maschine erhielt ihren Namen nach den Bezeichnungen ihrer vier prinzipiellen Register, die jeweils einen S-Ausdruck (bzw. einen Zeiger darauf) enthalten.

s: Der **Stack** wird verwendet zur Speicherung von Zwischenergebnissen bei der Auswertung eines Ausdrucks.

e: Die **Umgebungskomponente** (*environment*) enthält die Werte, die während der Auswertung an Variablen gebunden werden.

c: Die **Steuerliste** (*control list*) beinhaltet das gerade ausgeführte Maschinenprogramm.

d: Der **Dump** wird als ein Stack verwendet, um beim Aufruf einer neuen Funktion die Werte der anderen drei Register zu sichern.

Zu Beginn der Programmausführung werden die Register initialisiert mit

s	:=	NIL	e	:=	NIL
c	:=	⟨Programm⟩	d	:=	NIL

Bei der Analyse des Programms arbeitet die SECD-Maschine in solcher Weise, daß sie, entsprechend der Semantik der angetroffenen Maschineninstruktion, in einen anderen Zustand übergeht. Letztendlich wird der Wert, der sich aus der Anwendung der durch das Programm repräsentierten Funktion auf die Argumente ergibt, berechnet und auf der Spitze des Stacks abgelegt.

Der Zustand der Maschine wird durch die Angabe der Registerinhalte vollständig beschrieben. Daher können wir jede Maschineninstruktion durch die Angabe des Zustandes vor und nach der Ausführung wie folgt charakterisieren.

$$s\ e\ c\ d \quad \longrightarrow \quad s'\ e'\ c'\ d'$$

Die vier S-Ausdrücke links vom Pfeil geben den Zustand vor, die rechts vom Pfeil den Zustand nach der Ausführung an. Wollen wir ausdrücken, daß sich ein Registerinhalt nicht geändert hat, steht auf beiden Seiten der gleiche Buchstabe.

Die Maschine ist so organisiert, daß das Programm im Steuerregister immer eine Liste mit dem Befehlscode als erstem Element ist. Der Befehlscode bestimmt, welcher Zustandsübergang aktiviert wird. Die Maschine führt ihr Programm aus, indem sie durch die Folge von Übergängen, die von den Instruktionen im Steuerregister ausgelöst werden, geht. Die Abarbeitung ist beendet, wenn ein STOP-Befehl erkannt wird. Formal läßt sich dieser durch den Übergang

$$s\ e\ (\text{STOP})\ d \quad \longrightarrow \quad s\ e\ (\text{STOP})\ d$$

beschreiben, der, einmal angetroffen, die weitere Programmabarbeitung unterdrückt.

Die SECD-Maschine kennt außerdem die folgenden Befehle mit den folgenden Mnemonics.

LD: Laden eines Variablenwertes aus der Umgebung an die Spitze des Stack.

LDC: Laden einer Konstanten an die Stack-Spitze.

AP: Anwenden einer Funktion.

DUM: Erzeugen einer DUMMY-Umgebung für **letrec**-Ausdrücke.

RAP: Rekursive Funktionsanwendung.

SEL: Auswahl einer Unter-Steuerliste entsprechend einer Bedingung.

JOIN: Zurückkehren zur Hauptsteuerung nach Abarbeitung einer Unter-Steuerliste.

CAR , CDR: Bestimmen des Kopfes bzw. Schwanzes des gepunkteten Paares an der Stack-Spitze.

ATOM: Anwendung des **atom**-Prädikats auf das Objekt an der Stack-Spitze.

CONS: Bilden eines gepunkteten Paares aus den beiden obersten Stackelementen.

ADD , SUB, MUL, DIV, REM, LEQ, EQ: Ausführen der entsprechen arithmetischen Operation bzw. des "$\leq$"- oder "="-Prädikats auf die beiden obersten Stack-Elemente.

Im folgenden wollen wir uns die Zustandsübergänge für jeden dieser Befehle ansehen. Zuerst geben wir die Übergänge für die arithmetischen Operationen an. Im Falle der Addition erhalten wir

```
(a b.s) e (ADD.c) d  --->  (b+a.s) e c d
```

Für die Repräsentation der S-Ausdrücke verwenden wir die in *Lisp* übliche Schreibweise. Dabei machen wir von der Punktnotation Gebrauch, um die ersten Stack-Elemente zu bezeichnen; nach dem Punkt steht die Restliste. In unserem Fall bedeutet (a b.s), daß der Stack mindestens zwei Elemente hat, die nach der Operation durch deren Summe ersetzt sind.

Die Regeln für die anderen arithmetischen Operationen sehen genauso aus wie die der Addition.

```
(a b.s) e (SUB.c) d  --->  (b-a.s)     e c d
(a b.s) e (MUL.c) d  --->  (b*a.s)     e c d
(a b.s) e (DIV.c) d  --->  (b/a.s)     e c d
(a b.s) e (REM.c) d  --->  (b rem a.s) e c d
(a b.s) e (EQ.c)  d  --->  (b=a.s)     e c d
(a b.s) e (LEQ.c) d  --->  (b<=a.s)    e c d
```

Durch die letzten beiden Übergänge werden die beiden obersten Stackelemente durch eines der Atome T oder F, die für „wahr“ bzw. „falsch“ stehen, ersetzt.

Wie wir schon in der Befehlsliste gesehen haben, gibt es Maschinenbefehle für die drei Operationen über S-Ausdrücke, Kopf, Schwanz und CONS. Die Übergänge lauten.

```
(a b.s)   e (CONS.c) d  --->  ((a.b).s) e c d
((a.b).s) e (CAR.c)  d  --->  (a.s)     e c d
((a.b).s) e (CDR.c)  d  --->  (b.s)     e c d
```

Das Prädikat, das testet, ob das oberste Stackelement atomar ist oder nicht, hat den Übergang

```
(a.s) e (ATOM.c) d  --->  (t.s) e c d
```

wobei t = T ist, wenn a ein Atom ist, und t = F sonst.

Die restlichen Instruktionen sind spezifisch für die Struktur des erweiterten Lambda-Kalküls. Als erstes müssen wir unserer SECD-Maschine erlauben, auf die Variablenwerte zuzugreifen. Wir beschreiben daher zunächst die Struktur der Umgebungskomponente und die Beziehung dieser Struktur zum Gültigkeitsbereich der Variablenwerte.

Die Umgebungskomponente ist eine Wertliste, und zwar eine Liste von Listen aus Variablenwerten; z. B. ist

```
((3 17) (A B) (C))
```

eine mögliche Umgebung. Auf die Umgebung wird mit dem LD-Befehl, der ein Paar von Zahlen als Operand hat, zugegriffen. Die Sublisten der Umgebungskomponente sind mit 0, 1, ... numeriert, und jedes Element einer jeden Subliste hat seinerseits eine Nummer. Das Indexpaar (i, j) wählt das j-te Element der i-ten Subliste aus; z. B. führt das Paar (0.1) in der obigen Umgebung zum Wert 17. Jede Variable wird implizit mit einer Position in der Umgebungskomponente assoziiert. Jede Subliste entspricht einer Menge von Variablendefinitionen, die bei der Aktivierung einer Funktion oder eines **let(rec)**-Ausdruckes als neue Bindungen eingerichtet wurde, wobei die erste Subliste (mit der Nummer 0) die zuletzt erzeugten Bindungen beinhaltet.

Es gibt in der SECD-Maschine zwei Instruktionen, die für die Implementation des bedingten Ausdrucks benötigt werden. SEL wählt eine Subliste der Steuerliste in Abhängigkeit vom Wert an der Stack-Spitze aus; und JOIN wird verwendet, um zur Hauptsteuerung zurückzukehren. Wir erwarten, daß die Steuerliste die folgende Form hat

$$(\ldots \text{SEL} \quad \underbrace{(\ldots \text{JOIN})}_{c_T} \quad \underbrace{(\ldots \text{JOIN})}_{c_F} \quad \ldots)$$

und daß entweder c_T oder c_F ausgeführt werden, je nachdem, ob SEL den Wert T oder F an der Stack-Spitze findet. Nachdem eine der beiden Sublisten ausgeführt ist, veranlaßt JOIN die Rückkehr zu c. Die entsprechenden Übergänge haben demnach die Gestalt

```
(x.s) e (SEL cT cF.c) d      --->  s e cX (c.d)
s     e (JOIN)        (c.d)  --->  s e c  d
```

Wir sehen hier, daß SEL den Wert x an der Stack-Spitze untersucht und entweder cT oder cF auswählt, je nachdem, ob x = T oder x = F ist. Der Rest der Steuerliste c wird an der Dump-Spitze abgelegt. Ist die angewählte Subliste wohlgeformt in dem Sinne, daß sie den Dump so hinterläßt, wie sie ihn vorgefunden hat, stellt JOIN die Restliste der Steuerliste wieder zurück.

Die Instruktionen, die für die Auswertung von Funktionsrufen verwendet werden, sind komplexer. In der SECD-Maschine wird ein Funktionsobjekt als Closure dargestellt, die mit dem Befehl LDF erzeugt wird. Eine Closure ist hier ein Paar, das neben dem Code für den Lambda-Ausdruck den Wert der Umgebungskomponente beinhaltet, da unsere SECD-Maschine keine Assoziationsliste zur Darstellung der Umgebungen benutzt. Der LDF-Übergang lautet deshalb

```
s e (LDF c'.c) d  --->  ((c'.e).s) e c d
```

Da sich der Wert einer Variablen in einer reinen funktionalen Sprache nicht ändern darf, reicht es aus, einen Zeiger auf die aktuelle Umgebung in der Closure zu speichern. Sonst müßte man den Inhalt der Umgebungskomponente in die Closure kopieren.

Im allgemeinen wird die Closure nicht sofort nach ihrer Erzeugung angewendet, sondern wandert in die Umgebung, um dann öfters mit LD geladen und mit AP aktiviert zu werden.

Die AP-Instruktion ist die einzige Möglichkeit, um die Umgebungskomponente zu verändern, d. h. um Werte in ihr zu etablieren. Das bedeutet, daß einer Variablen nur einmal ein Wert gegeben werden kann, und zwar beim Funktionsaufruf oder beim Eintritt in einen **let(rec)**-Ausdruck. Diese Verfahrensweise ist ein Wesensmerkmal der reinen funktionalen Programmierung.

Wenn eine mit LDF definierte Funktion durch AP auf ihre Argumente angewendet wird, werden die Werte dieser Argumente als Subliste in der aus der Closure installierten Umgebung plaziert und sind daher über LD's vom Code der Funktion aus erreichbar. Außerdem installiert AP den Code der Funktion in der Steuerliste. Der Dump wird zur Sicherung der alten Registerinhalte verwendet. Die Funktionsabarbeitung beginnt mit dem leeren Stack.

```
((c'.e') v.s) e (AP.c) d  --->  NIL (v.e') c' (s e c.d)
```

AP erwartet, daß die Stack-Spitze eine Closure $(c'.e')$ und das zweite Stack-Element eine Liste v der Werte der Parameter von der durch c' repräsentierten Funktion ist.

Trifft die SECD-Maschine im folgenden auf einen RTN-Befehl, wird der Zustand, der im Dump aufbewahrt ist, zurückgeschrieben. RTN bewirkt also die Rückkehr aus der mit AP gestarteten Funktion und sollte am Ende des Funktionscodes stehen. Der Übergang lautet

```
(x) e' (RTN) (s e c.d)  --->  (x.s) e c d
```

RTN erwartet, daß der Funktionscode wohlgeformt ist, also den Dump so zurückläßt, wie er vor der eigentlichen Abarbeitung des Codes war. Außerdem

erwartet RTN vom Stack, daß er nur ein einziges Element — den Wert der Funktion — enthält, das dem rufenden Kontext übergeben wird, indem es auf den zurückgestellten Stack s gestapelt wird.

Um die restlichen beiden SECD-Maschinenbefehle, nämlich DUM und RAP, zu definieren, brauchen wir die *Lisp*-Pseudofunktion **rplaca**(x,y), die den Kopf von x durch y ersetzt. Diese Pseudofunktion hat x zum Wert, wobei aber der Kopf von x durch y ersetzt wurde. Es ist dabei nur erlaubt, den Kopf von x zu ersetzen, wenn dieser zuvor den speziellen Wert Ω („offen“) hatte. Ansonsten ist der Wert von **rplaca**(x,y) nicht definiert.

Wir benutzen **rplaca**, um rekursive **letrec**-Ausdrücke zu implementieren. Ein **letrec**-Ausdruck hat die Form

letrec
$$\begin{array}{lcl} v_1 & = & E_1 \\ \ldots & & \\ v_n & = & E_n \end{array}$$
in E

wobei die rechten Seiten E_i einer Definition in einer Umgebung ausgewertet werden müssen, in der die zu definierenden Variablen v_i verfügbar sind. Das sichern wir, indem wir eine Dummy-Umgebung erzeugen, in der die lokalen Definitionen offen sind, dann die Definitionen in der Dummy-Umgebung auswerten und zuletzt mit **rplaca** den offenen Teil der Umgebung durch die Werte der Definitionen ersetzen. Da die Umgebung bei der Auswertung der E_i noch offene Stellen hat, darf erst nach dem Ersetzen dieser offenen Stellen auf die v_i zugegriffen werden; die E_i dürfen daher nur Lambda-Abstraktionen sein, die zu Closures ausgewertet werden. Die Dummy-Umgebung wird also in jede dieser Closures, die durch die Definitionen erzeugt wurden, aufgenommen. Nach der Ersetzung der offenen Teile werden die Closures zirkular, d. h. sie enthalten eine Umgebung, die die Closures selbst enthält. Und das ist genau das, was wir mit den **letrec**'s beabsichtigen: rekursive und wechselseitig rekursive Definitionen.

Die DUM-Instruktion erzeugt eine Dummy-Umgebung mit Ω als erster Subliste. Daher ist bis zur Ersetzung von Ω jeder Zugriff auf diese Subliste undefiniert. Der Übergang für DUM ist gegeben durch

$$s\ e\ (\mathrm{DUM}.c)\ d \quad \longrightarrow \quad s\ (\Omega.e)\ c\ d$$

Der RAP-Befehl ist fast identisch zu AP, außer daß, anstatt die Liste der aktuellen Parameterwerte vorn an die Umgebung zu hängen, **rplaca** benutzt wird, um das Ω, das von DUM gesetzt wurde, zu ersetzen.

$$\begin{array}{lll} ((c'.e')\ v.s) & (\Omega.e) & (\mathrm{RAP}.c)\ d \\ & \longrightarrow & \mathrm{NIL}\ \mathrm{rplaca}(e', v)\ c'\ (s\ e\ c.d) \end{array}$$

Aus erster Sicht scheint dieser Übergang etwas ungewöhnlich zu sein. RAP wird immer in einem Zustand benutzt, in dem $e' = (\Omega.e)$ ist, d. h., die Closure an

s	e	DUM	
s	(Ω.e)	LDC	NIL
(NIL.s)	(Ω.e)	LDF	c_n
((c_n.(Ω.e)) NIL.s)	(Ω.e)	CONS	
(((c_n.(Ω.e)).NIL).s)	(Ω.e)	LDF	c_{n-1}
...			

der Spitze des Stack enthält eine Umgebung, die mit der aktuellen identisch ist. Andererseits benimmt sich RAP mehr wie AP. Es installiert den Code c' der Closure in der Steuerliste und die Umgebung e' in das Umgebungsregister, wobei der Kopf von e' durch die Liste v der Argumentwerte ersetzt ist. Stack, Umgebung und der Rest der Steuerliste werden auf den Dump gerettet. Im Falle der Umgebung, die ja mit DUM erweitert wurde, wird nur der Schwanz gesichert. Beim Vergleich mit dem Übergang für AP mit dem von RAP unter Beachtung der Tatsache, daß die Umgebung schon durch Ω erweitert wurde, sieht man leicht, daß beide miteinander vereinbar sind.

Die Verwendung dieser Befehle wird klarar, wenn wir den **letrec**-Ausdruck in die SECD-Maschinensprache übersetzen. Dabei gehen wir davon aus, daß

$$\begin{array}{ll} \textbf{letrec} & \\ \quad v_1 & = \quad \lambda \ldots . E_1 \\ \quad \ldots & \\ \quad v_n & = \quad \lambda \ldots . E_n \\ \textbf{in} \quad E & \end{array}$$

gleich dem Pseudo-Ausdruck

$$(\lambda* \quad (v_1 \ldots v_n).E) \quad \begin{array}{l} (\lambda \ldots . E_1) \\ \ldots \\ (\lambda \ldots . E_n)) \end{array}$$

ist, wobei das $\lambda*$ bedeuten soll, daß die v_i in den E_i verfügbar sind.

Das Gerüst des Maschinencodes für den **letrec**-Ausdruck ist dann folgendermaßen aufgebaut. Zuerst richten wir eine Dummy-Umgebung ein und laden die Closures, die die v_i definieren. Die c_i sind die Codes der Closures. Nach dem Laden der Closure, die v_1 definiert, erhalten wir an der Stack-Spitze eine Liste v der an die v_i gebundenen Closures.

$$v \quad = \quad (\,(c_1.(\Omega.e)) \ldots (c_n.(\Omega.e))\,)$$

Nun laden wir mit

$$\text{LDF} \quad c_E$$

die Closure des Ausdrucks E, wobei c_E dessen Code ist, und erhalten folgende Werte für Stack und Umgebungskomponente.

$$((c_E.(\Omega.e))\ v.s) \quad (\Omega.e)$$

Bei der darauffolgenden Ausführung des RAP-Befehls wird in der aktuellen Umgebung $(\Omega \,.\, e)$ der Kopf (also Ω) gegen die Liste v ersetzt. Da in den Closures der Definitionen nur Zeiger auf die aktuelle (nun veränderte) Umgebung stehen, werden die v_i innerhalb aller Definitionen (sowie natürlich im Ausdruck E) über ihre Position in der Umgebungskomponente sichtbar.

Ist eine Neubindung der Variablen erlaubt, muß in jeder Closure eine Kopie der aktuellen Umgebung stehen, um eine statische Variablenbindung zu erzielen.

Am Ende des nach dem RAP-Befehl abgearbeiteten Codes c_E, der wohlgeformt sein muß, steht ein RTN-Befehl, der die Maschinenregister zurückstellt und den Wert des Ausdrucks E (und damit des **letrec**-Ausdrucks) an der Stack-Spitze hinterläßt.

4.4 Verzögerte Auswertung mit der SECD-Maschine

Der Prozeß der verzögerten Auswertung ist am leichtesten zu erklären, wenn wir ihn explizit machen. Dazu führen wir neue Ausdrücke in unseren erweiterten Lambda-Kalkül ein, die das Verzögern sowie das Veranlassen der Auswertung eines Ausdrucks erlauben. Ist E ein wohlgeformter Ausdruck, dann ist auch (**delay** E) ein wohlgeformter Ausdruck. Der Wert von (**delay** E) ist ein Closure-ähnliches Objekt, das wir *Rezept* nennen. Ein Rezept enthält neben dem unausgewerteten Ausdruck die Umgebung, die für die Auswertung benötigt wird. Wenn E' ein wohlgeformter Ausdruck ist, so ist auch (**force** E') ein wohlgeformter Ausdruck. Der Wert von E' muß ein Rezept sein; dann ist der Wert von (**force** E') der Wert des Ausdrucks, der durch das Rezept eingekapselt wird.

Daher gilt für alle wohlgeformten Ausdrücke E, daß der Ausdruck (**force** (**delay** E)) immer denselben Wert wie E hat.

Die Erweiterung des Lambda-Kalküls mit **delay** und **force** ist eigentlich nicht notwendig, wenn man den Wert von **delay** E als die parameterlose Funktion $\lambda\,() \,.\, E$ betrachtet. Die Überführung eines Ausdrucks E in eine parameterlose Funktion verzögert dessen Auswertung. Erst der Aufruf dieser Funktion durch **force** veranlaßt die Auswertung.

Diese Implementation von **delay** und **force** ist uneffektiv, wenn die durch **delay** erzeugte parameterlose Funktion mehrmals mit **force** aufgerufen wird. Das ist jedoch nicht nötig, da die Auswertung von $\lambda\,() \,.\, E$ immer denselben Wert liefert. Deshalb ist die folgende Verfahrensweise von Vorteil. Wenn im Laufe der Berechnung die Funktion $\lambda\,() \,.\, E$ gerufen wird und erfolgreich endete, ersetzen wir danach die Closure, die den Wert von (**delay** E) repräsentiert, durch den Wert, den wir erhalten haben. Dadurch können nachfolgende (**force** E) direkt auf den Wert zugreifen, ohne die Closure noch einmal neu auszuwerten.

Um die beschriebene Verfahrensweise auf der SECD-Maschine ausführen zu können, müssen wir sie erweitern. Dazu brauchen wir drei neue SECD-

Maschinenbefehle, und zwar LDE (lade Ausdruck), AP0 (Aufruf einer parameterlosen Funktion) und UPD (Aktualisieren und Rückkehr aus der parameterlosen Funktion). Die Ausdrücke (**delay** E) und (**force** E') werden wie folgt compiliert.

$$\begin{array}{lcl} \textbf{delay}\ E & \longrightarrow & (\text{LDE}\ (E\ \text{UPD})) \\ \textbf{force}\ E' & \longrightarrow & (E'\ \text{AP0}) \end{array}$$

(**Delay** E) wird in eine Steuerliste der Form (LDE c) übersetzt, wobei c die bei der Compilierung von E erzeugte Steuerliste ist. Das letzte Element von c ist ein UPD-Befehl. Der Code für (**force** E') ist einfach die Liste der Instruktionen für E', gefolgt vom Befehl AP0.

Zur Vervollständigung unserer Definition müssen wir die SECD-Maschinenübergänge für die neuen Befehle angeben. Um das zu tun, definieren wir einen neuen Strukturtyp. Da wir einen verzögerten Ausdruck durch ein Rezept repräsentieren, das nach der Auswertung aktualisiert wird, kennzeichnen wir in diesem Rezept, ob es bereits ausgewertet ist oder nicht. Eine gewöhnliche Closure wird durch ein Paar (c.e) repräsentiert; für die Rezepte nutzen wir ein ähnliches Objekt. Ein Rezept wird mit

$$[F\ \ (c.e)]$$

bezeichnet, wenn es noch nicht ausgewertet ist. Dabei ist F die Markierung und $(c.e)$ der Closure-Teil. Ein bereits ausgewertetes Rezept bezeichnen wir mit

$$[T\ \ x]$$

wobei x der Wert und T die Markierung dafür ist, daß das Rezept ausgewertet ist.

Damit ist der Zustandsübergang für den LDE-Befehl gegeben.

```
s e (LDE c.c') d  ---> ([F (c.e)].s) e c' d
```

Wir sehen, daß LDE das Rezept $[F\ \ (c.e)]$ erzeugt und an die Spitze des Stack befördert. Die Abarbeitung setzt dann mit dem Rest c' des Programms fort, die Auswertung von c wird verzögert.

Als nächstes betrachten wir den Übergang für den AP0-Befehl, der ein Rezept an der Stack-Spitze erwartet. Dieses Rezept wurde entweder schon ausgewertet oder nicht. Wir behandeln jeden Fall getrennt.

```
([T x].s) e (AP0.c) d  --->  (x.s) e c d
```

Wenn das Rezept bereits ausgewertet war, nehmen wir einfach seinen Wert. War es noch nicht ausgewertet, müssen wir den Beginn der Auswertung ermöglichen.

```
([F (c.e)].s) e' (AP0.c') d
   --->  NIL e c (([F (c.e)].s) e' c'.d)
```

Das ist etwas kompliziert. Wir haben den gesamten Stack, einschließlich des Rezepts, zusammen mit der Umgebungskomponente und dem Rest der Steuerliste c' auf den Dump gerettet. Der Grund für die Sicherung des Rezepts besteht darin, das Rezept verfügbar zu haben, wenn wir es aktualisieren. Diese Situation entsteht, wenn wir die letzte Instruktion UPD der im Rezept eingebetteten Steuerliste erreicht haben. Der entsprechende Übergang sieht wie folgt aus.

```
(x) e (UPD) (([F (c.e)].s) e' c'.d)  --->  (x.s) e' c' d
und                          [F (c.e)]  --->  [T x]
```

Der Übergang selbst ist einfach; er zeigt an, daß, wenn die Abarbeitung von c beendet ist, der Wert x an den rufenden Kontext zurückgegeben und die Abarbeitung in diesem Kontext wieder aufgenommen wird; genauso, als wenn RTN benutzt wurde. Unser UPD-Übergang hat jedoch den Seiteneffekt des Aktualisierens des Rezeptinhalts, so daß dessen Flag nun T ist, was anzeigt, daß das Rezept ausgewertet ist, und daß der Wert x anstelle der Closure $(c.e)$ gespeichert wird. Rezepte werden normalerweise nicht kopiert, nur Zeiger auf sie, und so steht das symbolische Objekt $[F\ (c.e)]$ im obigen Übergang für alle Vorkommen (praktisch aber das einzige Vorkommen) des Rezepts. Daher beeinflußt dieses Aktualisieren als Seiteneffekt alle nachfolgenden Zugriffe auf das Rezept.

4.5 Die Funktionale Abstrakte Maschine

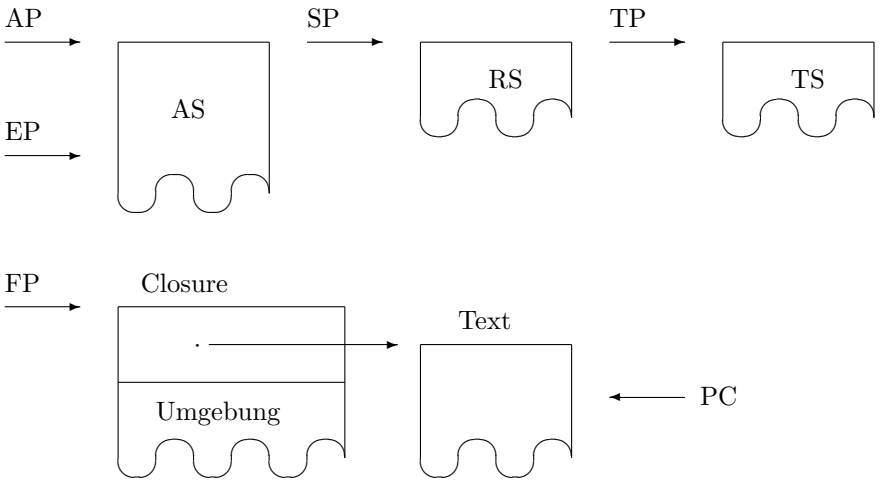
In den letzten beiden Abschnitten haben wir gesehen, daß die SECD-Maschine ein allgemeines Konzept der Implementation funktionaler Sprachen ist. Eine Konkretisierung dieses Konzepts führt uns zur Funktionalen Abstrakten Maschine. Die FAM ist eine Stack-Maschine, die für die Abarbeitung von funktionalen (speziell *ML*-) Programmen auf Rechnern mit großem Adreßraum entwickelt wurde. Sie kann als SECD-Maschine betrachtet werden, die dahingehend optimiert wurde, schnelle Funktionsanwendungen zu erlauben und richtige Stacks (anstelle verketteter Listen) zu benutzen.

Die FAM wurde von L. Cardelli entwickelt. Da uns der Originalartikel nicht zur Verfügung stand, haben wir den Überblick über diese Maschine dem Artikel [Cardelli 1984] entnommen.

Die Maschine unterstützt Funktionsobjekte (dynamisch erzeugte Closures), Mustererkennung, Schwanzrekursion und bestimmte Seiteneffekte. Alles, was die Funktionsanwendung langsamer macht, wurde vermieden. Die FAM-Instruktionen sollten nicht interpretiert, sondern in den Maschinencode des verwendeten Rechners übersetzt werden. Daher wurde auf optimierte Spezial-Operationen verzichtet.

Aus Effizienzgründen unterstützt die FAM keine Typprüfung zur Laufzeit und ist daher nicht typsicher. Wir müssen also die Typprüfung in der Quellsprache vornehmen, um die korrekte Anwendung der Maschinenoperationen zu garantieren.

Der Zustand der FAM ist durch sechs Zeiger mit ihren Bezeichnungen und eine Menge von Speicherplätzen eindeutig bestimmt. Die Zeiger sind der Argument-Pointer, der Frame-Pointer, der Stack-Pointer, der Trap-Pointer, der Programmzähler und der Umgebungszeiger. Sie zeigen auf drei unabhängige Stacks und, direkt oder indirekt, auf den Daten-Heap. Der Speicher achtet auf Seiteneffekte im Heap und beinhaltet das Dateisystem.



Der Argument-Pointer (AP) zeigt auf die Spitze des Argument-Stacks (AS), wohin Argumente, die an Funktionen weitergegeben werden, geladen und wo Funktionswerte zurückgelassen werden. Dieser Stack wird auch benutzt, um lokale Werte zu speichern.

Der Frame-Pointer (FP) zeigt auf die aktuelle Closure (bzw. das aktuelle Frame, FR), die den Text des gerade ausgeführten Programms und eine Umgebung für die freien Variablen dieses Programms enthält.

Der Programmzähler (PC) zeigt auf das Programm, das ausgeführt werden soll (PR) und das ein Teil der aktuellen Closure ist.

Der Stack-Pointer (SP) verweist auf die Spitze des Rückkehr-Stacks (RS), auf den Programmzähler und Frame-Pointer für die Dauer von Funktionsaufrufen gerettet werden.

Der Trap-Pointer (TP) zeigt auf den Trap-Stack (TS), wo Trap-Frames gespeichert sind. Ein Trap-Frame ist ein *Record* mit dem Maschinenzustand, das verwendet werden kann, um einen früheren Zustand wiederherzustellen (die Seiteneffekte im Heap werden aber nicht rückgängig gemacht).

Der Umgebungszeiger (EP) zeigt auch auf den Argument-Stack und definiert

die *Toplevel*-Umgebung zur Verwendung in interaktiven Systemen. Zu Beginn der Ausführung ist EP gleich dem Argument-Pointer, aber normalerweise wächst er infolge von *Toplevel*-Definitionen.

Die operationale Semantik dieser Maschine ist durch Zustandsübergänge gegeben. Einen Zustand repräsentieren wir durch das Tupel

```
AS  RS  FR  PR  TS  ES  M
```

M ist der Speicher. Das Programm PR (auf das PC zeigt) besteht aus einer Folge von abstrakten Maschineninstruktionen. Jede dieser Operationen bewirkt einen Zustandsübergang.

Closures werden erzeugt, indem die Werte der freien Variablen und der Funktionstext auf den Argument-Stack gebracht und von da aus in eine neu generierte Closure-Zelle übertragen werden. Closures für (wechselseitig) rekursive Funktionen können Zyklen enthalten und werden, wie bei der SECD-Maschine, in zwei Schritten erzeugt: Zuerst erfolgt der Aufbau von Dummy-Closures für eine Menge von wechselseitig rekursiven Funktionen, und danach entstehen die rekursiven Closures durch Füllen der Dummy-Closures.

Die Funktionsanwendung ist in drei Operationen aufgeteilt: **SaveFrame** zum Sichern der rufenden Closure auf den RS, **ApplFrame** zum Retten des rufenden Programmzählers auf den RS und zum Aktivieren der gerufenen Closure, die sich an der Argument-Stack-Spitze befindet, indem der FP auf die Closure und der PC auf ihren Eintrittspunkt gesetzt wird, sowie **RestFrame** zum Zurückstellen der rufenden Closure vom RS. **SaveFrame** und **RestFrame** sind zueinander invers und heben sich daher in mehrfachen (*curried*) Anwendungen auf. Die gerufene Closure verwendet **Return**, um den rufenden Programmzähler zurückzuspeichern und um zur rufenden Closure zurückzukehren (wo normalerweise ein **RestFrame** ausgeführt wird).

Des weiteren kennt die FAM Befehle zur Datenübertragung zwischen dem Argument-Stack und den Datenzellen, bedingte und unbedingte Sprünge, Befehle zur Ausnahme- und Fehlerbehandlung sowie primitive Operationen (wie z. B. die Addition).

Kapitel 5

Graphenreduktion

Bis jetzt nutzten wir eine Umgebung, um die Bindungen der Variablen eines Ausdrucks zu speichern. Zwei Bezüge auf die gleiche Variable veranlaßten den Abruf derselben Bindung aus der Umgebung; auf diese Weise ersparten wir uns, die Bindung zu duplizieren. Eine zweite Methode, eine solche Teilung zu erzielen, besteht darin, eine graphische Repräsentation eines Ausdrucks so zu wählen, daß mehrfache Bezüge zum gleichen Argumentausdruck durch mehrfache Kanten zu einem Argumentgraphen repräsentiert werden.

Den Prozeß der Graphenreduktion stellen wir uns vereinfacht so vor. Wir gehen davon aus, daß wir unser funktionales Programm in den Lambda-Kalkül übersetzt haben. Das abzuarbeitende Programm ist dann intern als Graph gespeichert. Der Reduktionsprozeß transformiert diesen Graphen, indem sukzessive Wurzeln von Teilausdrücken durch die entsprechenden Werte ersetzt werden. Die Reduktion ist beendet, wenn sich der Graph in einer Normalform befindet. Die Normalform ist dann der Wert eines Ausdrucks — und unser Programm ist auch nichts weiter als ein auszuwertender Ausdruck.

Die Graphenreduktion eignet sich besonders für Sprachen mit verzögerter Auswertung. Wir lassen also die „unreinen“ und die strikten Sprachen von vornherein weg, zumal die Sprachen mit verzögerter Auswertung einen echten semantischen Fortschritt gegenüber den strikten Sprachen darstellen. Das hat folgende Gründe.

- Die verzögerte Auswertung gestattet einen freieren, „mathematischen“ Programmierstil, da man die Auswertungsreihenfolge der verschiedenen Ausdrücke nicht zu beachten braucht.
- Sie erlaubt potentiell unendliche Datenstrukturen als Werte (von denen natürlich in endlicher Zeit nur endliche Teilstücke bearbeitet werden können). Außerdem kann man ohne Gefahr an Funktionen Argumente übergeben, deren Auswertung nicht terminiert, da die Auswertung nur so weit vorangetrieben wird, wie wirklich notwendig ist.

- Sie gestattet interaktive Ein- und Ausgaben in einer funktionalen Sprache. Ein Programm kann z. B. auf seine Standard-Eingabe in Form einer verzögert ausgewerteten Liste von Zeichen zugreifen. Weiterhin produziert das Programm eine Liste von Zeichen als Ergebnis, deren Elemente über die Standard-Ausgabe ausgegeben werden, sobald sie berechnet sind. Dieses Konzept, erweitert um die Ein- und Ausgabe auf Dateien, wurde in Lazy-ML [Augustsson 1989] verwirklicht.

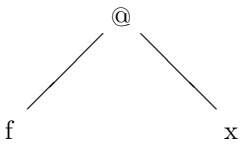
Wir haben fast die gesamten beiden Kapitel über die Graphenreduktion den Büchern [Peyton Jones 1987] und [Field 1988] entnommen, die gewissermaßen eine Pflichtlektüre für jeden, der sich mit der Implementation funktionaler Sprachen beschäftigt, sind.

Zuerst befassen wir uns mit der Repräsentation der Lambda-Ausdrücke, wie sie im Speicher des Rechners steht.

5.1 Programmrepräsentation

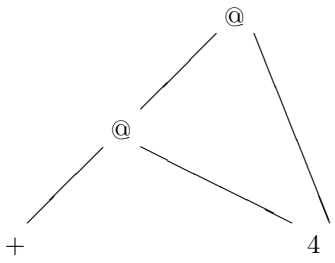
In jeder Implementation der Graphenreduktion ist der auszuwertende Ausdruck in Form eines gerichteten Graphen (eine Art Syntax-„Baum“) gespeichert, dessen Blätter Konstanten, eingebaute Funktionen oder Variablennamen sind.

Die Anwendung der Funktion f auf das Argument x repräsentieren wir wie folgt.



Das Zeichen „@“ ist die Markierung des Knotens und besagt, daß der Knoten eine Anwendung ist. Jede Kante führt nach unten vom Quell- zum Zielknoten.

Funktionen mit mehreren Argumenten behandeln wir durch *Curry*ing.

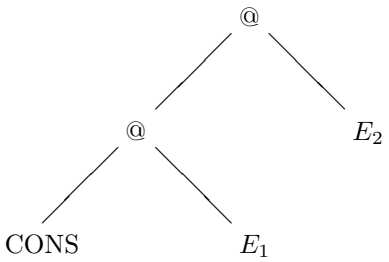


Dieser Graph repräsentiert den Ausdruck (+ 4 4), der die Funktion + bezeichnet, die auf 4 angewendet wird; das Ergebnis ist die Funktion (+ 4), die

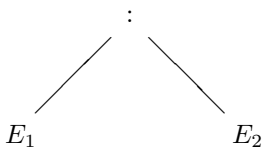
dann auf das Argument 4 angewendet wird. An diesem Beispiel sehen wir auch, wie Mehrfachbezüge auf den gleichen Teilausdruck realisiert sind.

Eine Lambda-Abstraktion ($\lambda\, x\,.$ Körper) stellen wir folgendermaßen dar.

Der Graph des Ausdrucks (CONS E_1 E_2) sieht so aus.



wobei E_1 und E_2 beliebige Ausdrücke sind. Das Resultat der Auswertung ist eine CONS-Zelle, die wie folgt repräsentiert ist.



wobei der Doppelpunkt den Knoten als eine CONS-Zelle markiert.

Die Bilder, die wir bis jetzt gesehen haben, sind ziemlich abstrakt. In einer typischen Implementation ist jeder Knoten durch ein kleines, zusammenhängendes Feld von Speicherplätzen, eine Zelle, repräsentiert. Jede Zelle enthält eine Markierung, die den Typ der Zelle (Anwendung, Abstraktion, Konstante usw.) angibt, und zwei oder mehrere Felder. Ein Feld kann einen Zeiger auf eine andere Zelle oder einen atomaren Datenwert beinhalten. Wir können eine Zelle wie folgt darstellen.

Markierung	Feld 1	$\cdots$	Feld n

Wir haben schon gesagt, daß eine Implementation einer funktionalen Sprache Konstruktorfunktionen unterstützen muß. Eine Konstruktorfunktion baut ein strukturiertes Datenobjekt auf, das einfach ein Aggregat von Werten mit einer Konstruktormarkierung, die den Konstruktor von den anderen Konstruktoren

desselben Typs unterscheidet, ist. Typischerweise ist die Konstruktormarkierung eine kleine ganze Zahl zwischen 1 und der Anzahl der Konstruktoren des Typs. Der Typ selbst braucht nicht mit abgespeichert werden, wenn wir das Programm bereits einer Typprüfung unterzogen haben.

5.2 Auswahl des nächsten reduzierbaren Ausdrucks

Nachdem der Graph eines funktionalen Programms in den Rechner geladen wurde, wird ein Auswerter (*evaluator*) gerufen, der den Graphen zu einer Normalform reduziert. Er tut das durch Ausführung wiederholter Reduktionen auf dem Graphen, die die beiden folgenden Aufgaben beinhalten.

- Auswahl des Ausdrucks, der reduziert werden soll (und kann).
- Reduktion.

In diesem Abschnitt beschäftigen wir uns mit dem ersten Teil, bevor wir im nächsten Abschnitt den zweiten Teil betrachten.

5.2.1 Verzögerte Auswertung

Jede Implementation der verzögerten Auswertung hat zwei Bestandteile.

1. Argumente von Funktionen sollten nur dann ausgewertet werden, wenn ihr Wert gebraucht wird, und nicht, wenn die Funktion angewendet wird.
2. Argumente sollten nur einmal ausgewertet werden, weitere Bezüge auf das Argument innerhalb der Funktion benutzen den Wert, der beim erstenmal berechnet wurde.

Wir müssen beide Teile unterstützen. Den zweiten Teil behandeln wir im nächsten Abschnitt, der erste wird durch die sogenannte Normalreduktion erledigt.

Normalreduktion (normal order reduction) besagt, daß der ganz links stehende, äußerste reduzierbare Ausdruck zuerst reduziert wird. Der ganz links stehende (*leftmost*) reduzierbare Ausdruck ist derjenige, dessen Lambda (oder primitiver Funktionsbezeichner) sich textlich links von allen anderen reduzierbaren Ausdrücken im Gesamtausdruck befindet. Ein äußerster reduzierbarer Ausdruck ist ein solcher, der nicht in einem anderen reduzierbaren Ausdruck enthalten ist.

5.2.2 Normalformen

Die Auswertung eines Ausdrucks, dessen Ergebnis eine CONS-Zelle ist, sollte nicht die Auswertung ihres Kopfes und ihres Schwanzes einschließen. Das heißt, daß wir die Reduktion stoppen, auch wenn noch reduzierbare Ausdrücke (nämlich Kopf und Schwanz) im Graphen verbleiben. Keiner dieser reduzierbaren Asudrücke wird von der Normalreduktion reduziert, bis der Gesamtausdruck zu einer CONS-Zelle ausgwertet wurde, weil bis dahin immer ein Ausdruck auf dem obersten Niveau vorhanden ist, den die Normalreduktion auswählt.

Wir verwenden daher die Normalreduktion, stoppen aber, wenn es auf dem obersten Niveau keinen reduzierbaren Ausdruck mehr gibt (auch, wenn im Graphen noch innere reduzierbare Ausdrücke verbleiben). Nach einer solchen Reduktion befindet sich unser Gesamtausdruck in schwacher Kopf-Normalform.

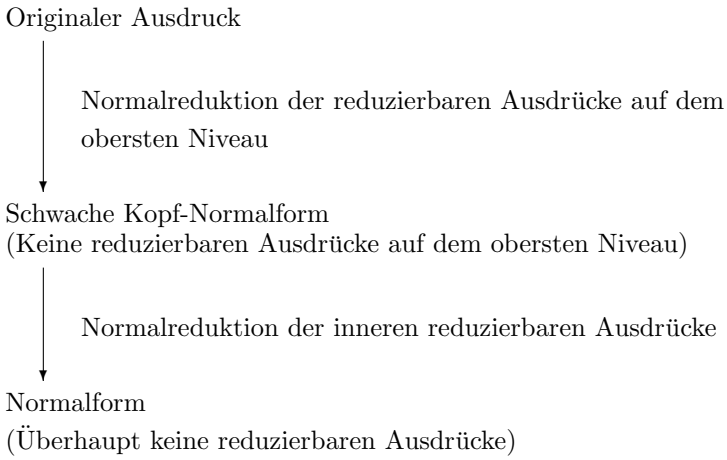
Ein Lambda-Ausdruck E ist genau dann in *schwacher Kopf-Normalform (weak head normal form)*, wenn

1. E ist eine Konstante
2. E ist ein Ausdruck $(\lambda\, x\, .\, E')$ für jedes E'
3. E ist von der Form $(P\ E_1\ \ldots\ E_n)$ für jede konstante Funktion P der Stelligkeit $k > n$.

Die dritte Regel besagt, daß jede partiell angewendete konstante Funktion auch eine schwache Kopf-Normalform ist. Das ist vernünftig, da wir einen Ausdruck wie z. B. $(*\ 3)$ nach einer Eta-Abstraktion durch den Ausdruck $(\lambda\, x\, .\, *\ \ 3\ \ x)$ ersetzen können, der in schwacher Kopf-Normalform ist.

Ein Ausdruck besitzt genau dann keine reduzierbaren Ausdrücke auf dem obersten Niveau, wenn er sich in schwacher Kopf-Normalform befindet.

Unsere Reduktionsordnung besteht nun darin, den reduzierbaren Ausdruck auf dem obersten iveau (das kann jeweils nur einer sein) zu reduzieren, bis wir die schwache Kopf-Normalform erreicht haben.



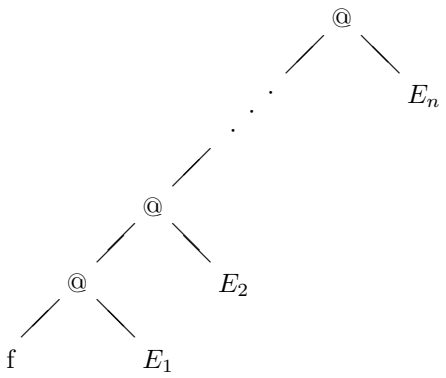
Wir verwenden die Normalreduktion und stoppen bei der schwachen Kopf-Normalform, und nicht erst bei der Normalform. Das ist ein wesentlicher Bestandteil der verzögerten Auswertung, da die Reduzierung zur Normalform unnötige Reduktionen riskiert.

Ein weiterer Vorteil der Reduzierung nur bis zur schwachen Kopf-Normalform besteht darin, daß wir keine Beta-Reduktionen in der Anwesenheit von freien Variablen ausführen müssen. Das einzige Mal, wo wir freie Variablen antreffen, ist, wenn wir „durch ein Lambda gehen“, da alle Bezüge zu der Variablen, die dieses Lambda einführt, nur im Körper dieser Lambda-Abstraktion frei sein kann. Da wir die Auswertung beim Lambda stoppen, vermeiden wir, in den Funktionskörper einzutreten, so daß wir überhaupt keine freien Variablen antreffen können. Daher tritt das Namenskonfliktproblem (*name-capture problem*), das entsteht, wenn eine freie Variable des Arguments einer Lambda-Abstraktion mit einem formalen Parameter im Lambda-Körper kollidiert, nicht auf.

5.2.3 Finden des nächsten reduzierbaren Ausdrucks auf dem obersten Niveau

Nachdem wir entschieden haben, die Normalreduktion nur der reduzierbaren Ausdrücke auf dem obersten Niveau zu implementieren, klären wir jetzt, wie wir in einem gegebenen Graphen den passenden reduzierbaren Ausdruck finden.

Unser Ausdruck kann nur die Form ($f\ E_1\ \ldots\ E_n$) haben, dessen Graph wie folgt aussieht.



Hier ist f ein Datenobjekt, eine eingebaute Funktion oder eine Lambda-Abstraktion, aber keine Anwendung (dann würden wir den Graphen nach unten verlängern). Es können null oder mehr Argumente E_i auftreten, die beliebig komplexe Ausdrücke sind. Nun gibt es verschiedene Möglichkeiten.

- f ist ein Datenobjekt, wie z. B. eine Zahl oder eine CONS-Zelle; in diesem Falle ist der Ausdruck bereits in schwacher Kopf-Normalform. Dann muß jedoch $n = 0$ sein, sonst würde ein Datenobjekt auf ein Argument angewendet werden! Das stellt einen Typfehler dar, der niemals auftritt, wenn vorher eine Typprüfung stattfand.

- f ist eine eingebaute Funktion mit k Argumenten. In diesem Falle prüfen wir, ob genügend Argumente vorhanden sind (d. h., ob $n \geq k$ ist).Wenn das so ist, ist $(f\ E_1\ \ldots\ Ek)$ der reduzierbare Ausdruck, den die Normalordnung auswählt. Ansonsten befindet sich der Ausdruck in schwacher Kopf-Normalform.

- f ist eine Lambda-Abstraktion. Wenn ein Argument verfügbar ist, ist $(f\ E_1)$ der nächste zu reduzierende Ausdruck. Gibt es kein Argument, ist der Ausdruck in schwacher Kopf-Normalform.

- f ist ein Variablenname. In diesem Falle tritt diese Variable im gesamten Ausdruck frei auf, was nicht sein darf und daher ein Fehler ist.

Um f zu finden, steigen wir in den linken Zweig jedes Anwendungs-Knotens, beginnend bei der Wurzel, ab, bis wir keinen Anwendungs-Knoten mehr vorfinden. Diese linksverzweigende Kette von Anwendungsknoten nennen wir das *Rückgrat* des Ausdrucks, und der Vorgang des Absteigens im Rückgrat ist das *Entfalten* des Rückgrats. Die *Wirbel* des Rückgrats sind die Anwendungs-Knoten, die beim Entfalten angetroffen werden, die *Rippen* sind die Argumente der Wirbel (die E_i), und die *Spitze* ist das äußerste untere Ende (f) des Rückgrats.

Es ist daher ziemlich einfach, den nächsten reduzierbaren Ausdruck für die Reduktion zu finden: Wir entfalten das Rückgrat, bis wir eine Funktion finden,

und steigen dann, in Abhängigkeit von dieser Funktion, wieder im Rückgrat auf, um die Wurzel des reduzierbaren Ausdrucks festzustellen.

5.3 Reduktionsregeln für Lambda-Ausdrücke

Die Ausführung einer Reduktion stellt eine lokale Transformation des Graphen, der den Ausdruck repräsentiert, dar. Der Prozeß der Reduktion modifiziert den Graphen so lange, bis eine Endform, das Ergebnis der Berechnung, erreicht ist.

Wie wir im vorigen Abschnitt gesehen haben, kann die Funktion an der Spitze des Rückgrates eine Lambda-Abstraktion oder eine eingebaute Funktion sein (dann hat der Graph einen reduzierbaren Ausdruck auf dem obersten Niveau). Wir betrachten jeden dieser beiden Fälle getrennt.

5.3.1 Reduktion einer Lambda-Abstraktion

Wenn der zu reduzierende Ausdruck eine Lambda-Abstraktion ist, die auf ein Argument angewendet wird, führen wir eine Beta-Reduktion auf dem Graphen aus. Das heißt, daß wir ein Beispiel (eine Instanz) des Körpers der Lambda-Abstraktion konstruieren müssen, in dem das Argument für alle freien Vorkommen des formalen Parameters eingesetzt ist. Dieser Prozeß wird auch als Instanziierung des Lambda-Körpers bezeichnet. Die Konstruktion einer neuen Instanz des Lambda-Körpers macht sich erforderlich, da die als Schablone dienende Lambda-Abstraktion aufgeteilt sein kann (d. h., daß im Gesamtgraphen noch weitere Kanten zu ihr führen).

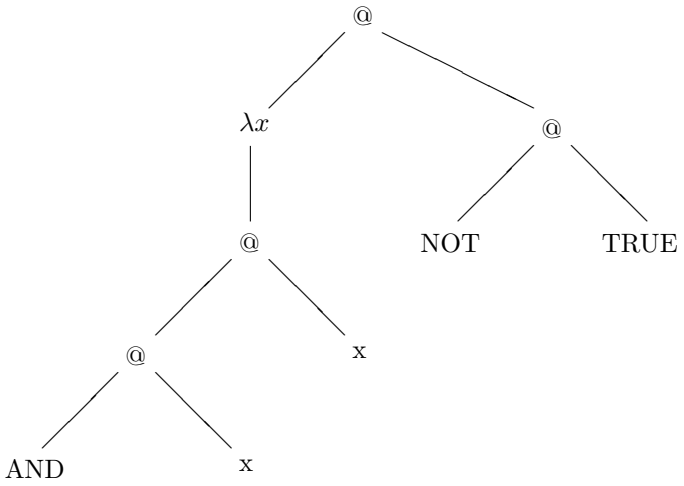
Bei der Substituierung des formalen Parameters durch das Argument könnten wir jedesmal das Argument kopieren. Das wäre ungünstig, weil

- das Argument ein sehr großer Ausdruck sein kann, so daß wir Speicherplatz für mehrfache Kopien desselben Objekts verschwenden

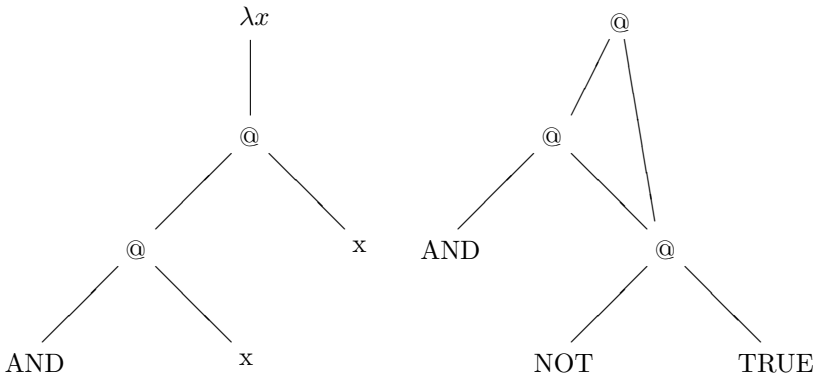
- das Argument reduzierbare Ausdrücke enthalten kann, so daß wir Zeit verschwenden, wenn wir alle Kopien getrennt reduzieren.

Beide Probleme umgehen wir, wenn wir die Bezüge auf den formalen Parameter durch Zeiger auf das Argument ersetzen. Dadurch wird das Argument aufgeteilt.

Die folgende Figur gibt ein Beispiel für den Reduktionsprozeß.



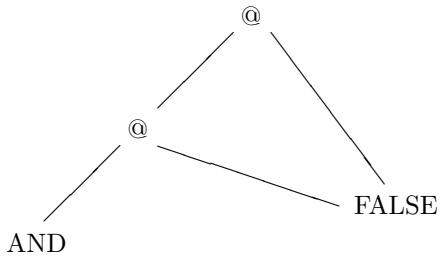
wird reduziert zu



Wenn wir die Teilung erfolgreich ausnutzen wollen, müssen wir nach der Reduktion den Graphen modifizieren, um das Ergebnis widerzuspiegeln. Damit erreichen wir, daß geteilte Ausdrücke nur einmal reduziert werden.

Die Modifikation des Graphen besteht einfach darin, daß wir die Wurzel des reduzierten Ausdrucks physisch mit der Wurzel des Ergebnisgraphen überschreiben.

Unser Beispielgraph wird also nach der Reduktion von (NOT TRUE) wie folgt aussehen.



NOT TRUE

Teile des reduzierten Ausdrucks (in unserem Falle der NOT- und der TRUE-Knoten) bleiben vom Überschreiben unbeeinflußt, werden aber vom betrachteten Teilgraphen abgekoppelt. Sie können nicht sofort freigegeben werden, da sie eventuell geteilt sein können. Sind sie es nicht, werden sie letztendlich vom *garbage collector* beseitigt.

Wir sagten, daß die verzögerte Auswertung zwei Bestandteile hat.

- Argumente von Funktionen sollten nur dann ausgewertet werden, wenn es erforderlich ist.
- Einmal ausgewertet, sollten sie nicht noch einmal ausgewertet werden.

Die Normalreduktion implementiert den ersten Bestandteil. Der zweite Teil wird durch die Kombination von zwei Dingen erreicht.

- Einsetzen von Zeigern auf das Argument, anstatt es zu kopieren.
- Das Aktualisieren der Wurzel des reduzierten Ausdrucks mit der Wurzel des Ergebnisses sichert, daß weitere Verwendungen des Arguments von der getanen Arbeit profitieren.

Diese Implementationsstrategie wird *träge Graphenreduktion (lazy graph reduction)* genannt.

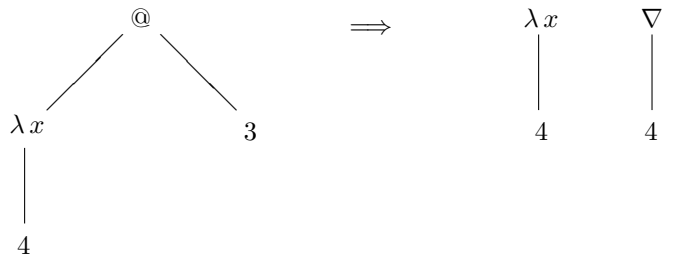
Das Überschreiben der Wurzel des reduzierten Ausdrucks funktioniert nicht so einfach, wenn das Ergebnis der Reduktion überhaupt keine Wurzelzelle hat. Betrachten wir den Ausdruck

$$(\lambda\, x\,.\, 4)\ \ 5$$

dessen Reduktion den Wert 4 liefert. In einer Implementation, in der Zahlen als Nicht-Zeiger repräsentiert sind, beanspruchen sie keine vollständige Zelle.

Wie können wir nun die Wurzel des reduzierten Ausdrucks mit einem solchen Objekt überschreiben? Wir führen einen neuen Zellentyp, die Verweis-Zelle, ein, deren einziges Feld einen solchen Nicht-Zeiger enthält.

Jetzt können wir die Reduktion ausführen



wobei wir das Zeichen ∇ benutzen, um Verweis-Knoten zu bezeichnen.

5.3.2 Reduzieren der eingebauten Funktionen

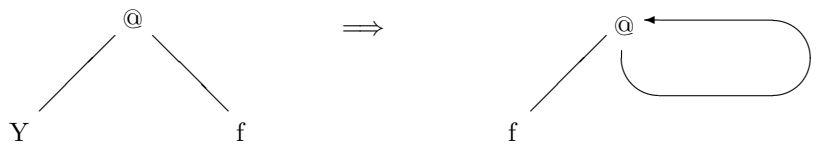
Wir nehmen an, daß unser zu reduzierender Ausdruck aus einer eingebauten Funktion besteht, die auf die korrekte Anzahl von Argumenten angewendet wird. Zuerst werten wir die Argumente, deren Werte gebraucht werden, durch rekursiven Aufruf des Auswerters aus. Dann kann die eingebaute Funktion ausgeführt werden, und das Ergebnis überschreibt die Wurzel des reduzierten Ausdrucks.

5.3.3 Implementation von Y

Der Fixpunkt-Kombinator **Y** kann direkt implementiert werden. Die Reduktionsregel für **Y** ist

$$\mathbf{Y}\ f \quad \longrightarrow \quad f\ (\mathbf{Y}\ f)$$

und kann so realisiert werden



Um zu sehen, daß das eine korrekte Implementation ist, betrachten wir die Reduktionsregel für **Y**. Auf der rechten Seite der Regel wird f auf (**Y** f) angewendet, doch das ist der originale Ausdruck; und so kann f auf die Wurzel des originalen Ausdrucks angewendet werden.

Auf diese Weise wird unser Graph zyklisch. Zyklische Graphen sind endliche Repräsentationen von unendlichen Objekten (z. B. rekursiven Funktionen und einigen unendlichen Datenstrukturen).

5.4 Superkombinatoren und Lambda-Lifting

Da die Instanziierung eines Lambda-Körpers die fundamentale Operation unserer Implementation ist, wollen wir nun betrachten, wie wir sie effizienter gestalten. Dazu werden wir unsere Lambda-Ausdrücke in eine Form bringen, in welcher die Lambda-Abstraktionen besonders leicht zu instanziieren sind. Diese speziellen Lambda-Abstraktionen heißen Superkombinatoren, und die Transformation wird „Lambda-Lifting“ genannt.

5.4.1 Die Idee der Compilation

Die Instanziierung des Körpers einer Lambda-Abstraktion, die durch rekursive Baum-Traversierung über dem Lambda-Körper ausgeführt wurde, ist ziemlich ineffizient, und zwar aus den folgenden Gründen.

- An jedem Knoten des Körpers muß die Instanziierungs-Operation eine Fallanalyse der Knotenmarkierung ausführen.
- Bei jedem Variablen-Knoten muß die Operation testen, ob der Knoten der formale Parameter ist; ein ähnlicher Test wird bei jedem Lambda-Knoten gemacht.
- Es werden auch neue Instanzen von Unterausdrücken, die keine freien Vorkommen des formalen Parameters enthalten, erzeugt, die jedoch geteilt werden könnten.

Eine effizientere Alternative dazu ist die *Compilation*. Durch die Compilation wird jeder Lambda-Körper mit einer festen Folge von Instruktionen assoziiert, die eine Instanz dieses Lambda-Körpers konstruiert. Die Instanziierungs-Operation besteht dann einfach aus dem Befolgen der mit dem Lambda-Körper verbundenen Instruktionenfolge.

Die Instruktionenfolge kann durch einen Compiler generiert werden und enthält implizit das Wissen über die Form des Körpers und die Vorkommen der formalen Parameter.

Unglücklicherweise eignen sich nicht alle Lambda-Abstraktionen für eine Compilation dieser Art. Betrachten wir z. B. die Lambda-Abstraktion

$$\lambda x\,.\,(\lambda y\,.\,-\;\;y\;\;x)$$

Wenn wir die Lambda-Abstraktion auf ein Argument, sagen wir 3, anwenden, instanziieren wir ihren Körper und erhalten eine neue Lambda-Abstraktion

$$\lambda y\,.\,-\;\;y\;\;3$$

Jede Anwendung der λx-Abstraktion auf ein anderes Argument erzeugt eine andere λy-Abstraktion. Damit können wir die Hoffnung, aus jeder Lambda-Abstraktion eine einzige feste Code-Folge zu erzeugen, begraben. Das Problem

besteht darin, daß x im Körper der λy-Abstraktion frei ist, so daß wir eine neue Instanz der λy-Abstraktion erzeugen müssen, wann immer x durch eine Anwendung der λx-Abstraktion neu gebunden wird. Nur im Falle der Lambda-Abstraktionen ohne freie Variablen verläuft die beschriebene Compilations-Methode problemlos.

Ein Ausweg besteht darin, der Code-Sequenz den Zugriff auf die Werte der freien Variablen zu erlauben. Diese Idee führt zur umgebungsbasierten Übersetzung, die wir bereits besprochen haben.

Hier wollen wir uns eine andere Möglichkeit, die Superkombinator-Graphenreduktion, ansehen, die keine Hinzunahme einer Umgebung zu unserem Modell der Graphenreduktion erfordert.

5.4.2 Lösung des Problems der freien Variablen

Wir lösen dieses Problem der freien Variablen, indem wir eine modifizierte Form der Beta-Reduktion verwenden, mit der man mehrere gewöhnliche Beta-Reduktionen auf einmal ausführen kann.

Betrachten wir unser Beispiel

$$\lambda x\,.\,\lambda y\,.\,-\;\;y\;\;x$$

und wenden es auf zwei Argumente an, z. B.

$$(\lambda x\,.\,\lambda y\,.\,-\;\;y\;\;x)\;\;\;3\;\;\;4$$

Unser bisherigen Graphenreduzierer würde diesen Ausdruck folgendermaßen bearbeiten.

$$\begin{aligned}(\lambda x\,.\,\lambda y\,.\,-\;\;y\;\;x)\;\;\;3\;\;4\;\;&\longrightarrow\;\;(\lambda y\,.\,-\;\;y\;\;3)\;\;\;4\\&\longrightarrow\;\;-\;\;4\;\;3\end{aligned}$$

Es spricht jedoch nichts dagegen, beide Reduktionen gleichzeitig auszuführen.

$$(\lambda x\,.\,\lambda y\,.\,-\;\;y\;\;x)\;\;\;3\;\;4\;\;\;\longrightarrow\;\;\;-\;\;4\;\;3$$

Die Multi-Argument-Reduktion hat die Konstruktion einer Instanz des Körpers $(-\;y\;x)$ zur Folge, in der freie Vorkommen von x durch 3 und freie Vorkommen von y durch 4 ersetzt sind.

Die folgenden Beobachtungen sind wichtig.

- Es wird einiges durch die gleichzeitige Ausführung der Reduktionen gewonnen. Erstens benötigen wir weniger Zwischenstrukturen, da das Zwischenergebnis der λx-Reduktion nicht konstruiert wird. Zweitens, und wichtiger, treten keine Probleme durch das freie Vorkommen von x in der λy-Abstraktion auf.

- Nichts wird verloren, wenn wir die $\lambda\, x$- und die $\lambda\, y$-Reduktion gleichzeitig vornehmen. Das Resultat der Ausführung der $\lambda\, x$-Reduktion ist eine $\lambda\, y$-Abstraktion; und es kann (unter Verwendung der Normalreduktion bis zur schwachen Kopf-Normalform) nichts mit der $\lambda\, y$-Abstraktion gemacht werden, bis ihr ein weiteres Argument gegeben wird. Daher können wir warten, bis beide Argumente vorhanden sind, und dann beide Reduktionen als eine ausführen. Das funktioniert sogar, wenn die Anwendung der $\lambda\, x$-Abstraktion auf ein einzelnes Argument geteilt ist.

Welche Sorte von Lambda-Abstraktionen sind nun für diese Multi-Argument-Reduktion geeignet? Einfach Lambda-Abstraktionen der Form $(\lambda\, x_1\, .\, \lambda\, x2\, \ldots\, \lambda\, x_n\, .\, E)$. Das motiviert die folgende Definition.

Ein *Superkombinator* \$S der Stelligkeit n ist ein Lambda-Ausdruck der Form

$$\lambda\, x_1\, .\, \lambda\, x_2\, \ldots\, \lambda\, x_n\, .\, E$$

wobei E keine Lambda-Abstraktion ist (das sichert, daß die „führenden Lambdas“ von 1 bis n numeriert sind), so daß

1. \$S hat keine freien Variablen
2. Jede Lambda-Abstraktion in E ist ein Superkombinator
3. $n \geq 0$, d. h. es brauchen überhaupt keine Lambdas aufzutreten.

Ein *reduzierbarer Ausdruck mit einem Superkombinator* ist die Anwendung eines Superkombinators auf n Argumente, wenn n seine Stelligkeit ist. Eine *Superkombinator-Reduktion* ersetzt einen reduzierbaren Ausdruck mit einem Superkombinator gegen eine Instanz des Superkombinator-Körpers, in der die freien Vorkommen der formalen Parameter durch die entsprechenden Argumente ersetzt sind.

Superkombinatoren der Stelligkeit von ungleich Null sind unsere Einheit der Compilation. Da sie keine freien Variablen besitzen, können wir aus ihnen eine feste Code-Sequenz erzeugen. Außerdem hat keine Lambda-Abstraktion im Superkombinator-Körper freie Variablen, sie brauchen also bei der Instanziierung nicht kopiert zu werden.

Es hört sich an, als wäre ein „Superkombinator“ eine spezielle Art „Kombinator“; und es ist wirklich so. Ein *Kombinator* ist ein Lambda-Ausdruck, der keine freien Variablen enthält. Ein Kombinator ist eine „reine“ Funktion in dem Sinne, daß der Wert eines Kombinators, angewendet auf Argumente, nur von den Werten dieser Argumente abhängt.

Nun hat aber ein durchschnittliches funktionales Programm nicht nur Lambda-Abstraktionen, die Superkombinatoren sind, aber wir können relativ

einfach unser Programm so transformieren, daß es nur noch Superkombinatoren enthält.

Wir werden unsere Superkombinatoren oft benennen. Diese Namen sind beliebig, da die Lambda-Abstraktionen anonym sind, und beginnen mit einem „$“. So könnten wir schreiben

$$\$\mathrm{XY} \quad = \quad \lambda x\,.\,\lambda y\,.\,-\;\; y\;\; x$$

doch um ihren speziellen Status zu betonen, schreiben wir die Definition so

$$\$\mathrm{XY}\;\; x\;\; y \quad = \quad -\;\; y\;\; x$$

Unsere Strategie ist nun, den zu compilierenden Lambda-Ausdruck in eine Menge von Superkombinator-Definitionen und einen auszuwertenden Ausdruck zu transformieren.

Ein entscheidender Punkt in der Definition eines Superkombinators ist, daß eine Superkombinator-Reduktion nur dann stattfindet, wenn alle Argumente verfügbar sind. Wir können die Superkombinator-Definitionen als eine Menge von Umformungsregeln (*rewriting rules*) betrachten. Eine Reduktion besteht aus der Umformung eines Ausdrucks, der auf die linke Seite einer Regel paßt, in ein Beispiel der entsprechenden rechten Seite. Solche Systeme heißen Termumformungssysteme (*term rewriting systems*).

5.4.3 Überführung von Lambda-Abstraktionen in Superkombinatoren

Wir betrachten folgendes Beispiel-Programm, in dem keine Lambda-Abstraktion ein Superkombinator ist.

$$(\lambda x\,.\,(\lambda y\,.\;\; (+\;\; x\;\; y)\;\; x)\;\; 4$$

(Über der Linie stehen später die Superkombinator-Definitionen). Nehmen wir uns zuerst die innerste Lambda-Abstraktion

$$\lambda y\,.\,+\;\; y\;\; x$$

Sie hat eine freie Variable und ist daher kein Superkombinator. Jedoch eine einfache Transformation macht aus ihr einen.

> *Mache jede freie Variable zu einem extra Parameter (wir bezeichnen das auch als Abstraktion der freien Variablen).*

Wir können jetzt

$$(\lambda y\,.\,+\;\; y\;\; x)$$

in

$$(\lambda x\,.\,\lambda y\,.\,+\;\; y\;\; x)\;\; x$$

transformieren (diese Operation ist eine Beta-Abstraktion). Um den Vorgang klarer zu machen, führen wir eine Alpha-Konversion auf der erhaltenen Lambda-Abstraktion aus und bekommen

$$(\lambda\, w\,.\,\lambda\, y\,.\,+\;\; y\;\; w)\;\; x$$

Das macht die Unterscheidung der beiden x in der vorigen Version deutlich. Nun ist unsere Lambda-Abstraktion ein Superkombinator, und nach der Transformation unseres Originalprogramms erhalten wir

$$(\lambda\, x\,.\,(\lambda w.\lambda\, y\,.\,+\;\; y\;\; w)\;\; x\;\; x)\;\; 4$$

Dann geben wir dem Superkombinator einen Namen, sagen wir \$Y.

$$\frac{\$Y\;\; w\;\; y\;\; =\;\; +\; y\; w}{(\lambda\, x\,.\,\$\mathrm{Y}\;\;\; \mathrm{x}\;\;\; \mathrm{x})\;\;\; 4}$$

Jetzt ist auch die $\lambda\, x$-Abstraktion ein Superkombinator, und wir bezeichnen ihn mit \$X.

$$\frac{\begin{array}{l}\$\mathrm{Y}\;\; \mathrm{w}\;\; \mathrm{y}\;\; =\;\; +\;\; \mathrm{y}\;\; \mathrm{w} \\ \$\mathrm{X}\;\; \mathrm{x}\;\; =\;\; \$\mathrm{Y}\;\; \mathrm{x}\;\; \mathrm{x}\end{array}}{\$\mathrm{X}\;\; 4}$$

Wir erhalten den folgenden Algorithmus.

SOLANGE noch Lambda-Abstraktionen vorhanden sind **MACHE**

- Wähle irgendeine Lambda-Abstraktion, die keine innere Lambda-Abstraktion in ihrem Körper hat.
- Mache alle freien Variablen zu extra Parametern.
- Benenne die Lambda-Abstraktion eindeutig.
- Ersetze das Vorkommen der Lambda-Abstraktion durch den Namen, angewendet auf die freien Variablen.
- Compiliere die Lambda-Abstraktion und assoziiere den Namen mit dem compilierten Code.

ENDE

Wir sehen, daß wir dadurch das Programm vergrößern, aber wir haben den Vorteil einfacherer Reduktionsregeln. Nach Beendigung des Algorithmus erhalten wir ein Programm der Form

... Superkombinator-Definitionen ...

E

Da der Ausdruck E keine freien Variablen enthalten darf — denn er ist der Ausdruck auf dem obersten Niveau —, können wir ihn in einen parameterlosen Superkombinator überführen und bekommen das endgültige Programm

... Superkombinator-Definitionen ...
$Prog = E

$Prog

Nach Johnsson [Johnsson 1987] bezeichnen wir diese Transformation als *Lambda-Lifting*, da alle Lambda-Abstraktionen auf das oberste Niveau gehoben werden.

5.5 Rekursive Superkombinatoren

Bis jetzt haben wir noch nicht erwähnt, wie unser Lambda-Lifter rekursive Definitionen behandeln soll. Ein Weg wäre die Transformation aller rekursiven Definitionen in nichtrekursive unter Benutzung des Fixpunkt-Kombinators **Y**. Das ist aber ineffizient und langsam, und zwar aus den folgenden Gründen.

- Es gibt keinen Grund, warum die Superkombinatoren nicht explizit rekursiv sein sollten, da sie Namen tragen und somit auf sich selbst verweisen können; z. B.

  $$\$F\ x \quad = \quad \$G\ (\$F\ (-\ x\ 1))\ 0$$

- Wenn wir $F mit Hilfe von **Y** nichtrekursiv machen, benötigen wir eine Hilfsfunktion.

  ```
  $F          =  Y  $F1
  $F1  F  x   =  $G  (F  (-  x  1))  0
  ```

  Definieren wir $F auf diese Weise, brauchen wir mehr Reduktionen als die explizit rekursive Version, da **Y** reduziert werden muß.

- Die Übersetzung von **letrec**-Ausdrücken mit wechselseitig rekursiven Definitionen haben wir vorgenommen, indem wir zuerst die Definitionen zu

einem Tupel vereinigten und diese resultierende Definition mit **Y** nichtrekursiv machten. Es ist nicht nur ärgerlich, zur Behandlung der wechselseitigen Rekursion Tupel einzuführen, sondern auch ineffizient, da die Tupel zuerst konstruiert und dann auseinandergenommen werden müssen.

Daher sind die explizit rekursiven Definitionen von Superkombinatoren die bessere Lösung. Wir beschreiben nun die Techniken zur Erlangung einer Menge von wechselseitig rekursiven Superkombinator-Definitionen ohne Benutzung von **Y**. Dabei setzen wir voraus, daß das funktionale Programm in eine durch **let**- und **letrec**-Konstrukte erweiterte Lambda-Notation übersetzt ist.

Nebenbei sehen wir, daß unsere Notation

$$\frac{\begin{array}{l}\$R\ x\ y\ =\ B1\\ \$S\ f\ =\ B2\\ \ldots\end{array}}{E}$$

äquivalent zu

letrec

$\$R = \lambda x.\lambda y.B1$

$\$S = \lambda f.B2$

$\ldots$

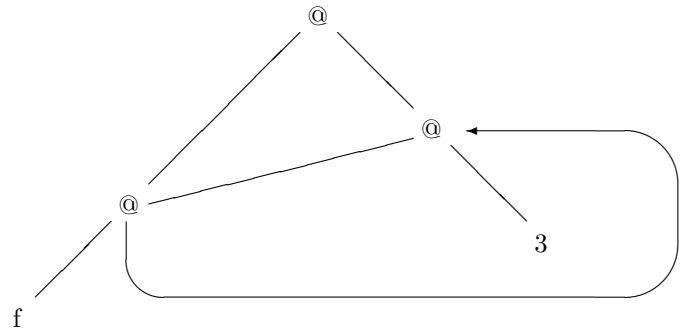
in

E

ist, so daß der Lambda-Lifting-Prozeß als eine Transformation innerhalb des erweiterten Lambda-Kalküls betrachtet werden kann.

5.5.1 Let's und letrec's in Superkombinator-Körpern

Betrachten wir den Graphen



Ausdrücke der Form $(f\ (g\ a)\ b)$ können den Zyklus im Graphen nicht beschreiben. Eine Lösung wäre die Benennung der Knoten (z. B. mit a, b und c von oben nach unten), so daß wir den Graphen wie folgt ausdrücken.

$$\begin{array}{lcl} a & = & c\ \ b \\ b & = & c\ \ 3 \\ c & = & f\ \ b \end{array}$$

Wenn wir annehmen, daß a die Wurzel unseres Graphen ist, erhalten wir als Äquivalent für diese Beschreibung den **letrec**-Ausdruck

letrec

a = c b
b = c 3
c = f b

in

a

Daraus folgt, daß wir die **letrec**-Ausdrücke als textliche Beschreibungen von zyklischen Graphen ansehen können. Ein **letrec** in einem Superkombinator-Körper kann daher als die Beschreibung eines graphischen Teils des Superkombinator-Körpers betrachtet werden. Wir sagen dann, daß ein solcher Superkombinator einen graphischen Körper hat.

Als Beispiel sehen wir uns die zyklische Version des **Y**-Kombinators an, dessen Körper ein Graph ist.

$$\begin{array}{lcl} \$Y\ f & = & \textbf{letrec}\ \text{yf} = f\ \text{yf} \\ & & \textbf{in}\ \text{yf} \end{array}$$

Wenn \$Y angewendet wird, fertigen wir eine Instanz des Graphen an, wobei wir freie Vorkommen des formalen Parameters f ersetzen. Bei der Instanziierung dürfen die Zyklen im Originalgraphen nicht verlorengehen.

In ähnlicher Weise betrachten wir **let**-Ausdrücke in einem Superkombinator-Körper als Beschreibungen für azyklische Graphen. Wir können daher Ausdrücke der Form

$$\textbf{let} \quad x \;=\; 3 \quad \textbf{in} \quad E$$

direkt beschreiben, die wir früher in

$$(\lambda\, x\,.\,E)\ 3$$

übersetzen mußten. Das direkte Vorkommen von **let**'s in Superkombinatoren spart eine explizite Reduktion der Lambda-Abstraktion, die in der zweiten Form auszuführen ist.

Wir sehen also, daß

- es ziemlich einfach ist, die Superkombinatoren so zu erweitern, daß sie Körper haben, die allgemeine Graphen sind
- graphische Superkombinator-Körper durch **letrec**'s (oder **let**'s für azyklische Körper) beschrieben werden können
- wir bei der Instanziierung der **let**'s und **letrec**'s einfach den Graphen, den der **let**- bzw. **letrec**-Ausdruck beschreibt, konstruieren
- wir durch die Verwendung graphischer Körper Reduktionen sparen können.

5.5.2 Lambda-Lifting in Anwesenheit von letrec's

Unser Lambda-Lifting-Algorithmus funktioniert wie vorher, **letrec**'s werden wie jeder andere Ausdruck behandelt. Jedoch wird das Lambda-Lifting nur auf Lambda-Abstraktionen, nicht aber auf **letrec**'s, angewendet. Einige Lambda-Abstraktionen haben **letrec**'s in ihrem Körper, die zu Superkombinatoren mit graphischen Körpern führen.

Wenn ein **letrec**-Ausdruck nicht in einer Lambda-Abstraktion eingeschlossen ist (d. h., wenn sie sich auf dem obersten Niveau befindet), können die Körper seiner Definitionen keine freien Variablen (außer den anderen im **letrec** definierten Variablen) enthalten, sie sind also Kombinatoren. Alles, was wir zu tun haben, ist deren Überführung in Superkombinatoren, indem wir alle inneren Lambdas durch Liften entfernen. Dabei beachten wir, daß die in sich auf dem höchsten Niveau befindenden **letrec**'s gebundenen Variablen nicht als freie Variablen abstrahiert werden, da wir keine Konstanten abstrahieren wollen.

Angenommen, wir möchten das Programm, das eine unendliche Liste von Einsen erzeugt, liften.

$$\begin{array}{rcl} \textbf{letrec} & x & = \quad \text{CONS}\ 1\ x \\ \textbf{in} & x & \end{array}$$

Der **letrec**-Ausdruck ist auf dem obersten Niveau und es gibt keine Lambda-Abstraktion, so daß x bereits ein Superkombinator ist. Wir erhalten

$$\frac{\$x \quad = \quad \text{CONS} \;\; 1 \;\; \$x}{\$x}$$

Als Beispiel einer rekursiven Funktion betrachten wir die Fakultäts-Funktion.

$$\begin{array}{l} \textbf{letrec} \quad fac \;=\; \lambda n \,.\, IF \quad (=\; n\; 0)\; 1 \\ \qquad\qquad\qquad\qquad\quad\;\; (*\; n\; (fac\; (-\; n\; 1))) \\ \textbf{in} \quad fac\; 4 \end{array}$$

Das **letrec** ist auf dem obersten Niveau, und der Körper der λn-Abstraktion enthält keine weitere Lambda-Abstraktion. Daher ist fac ein Superkombinator, wir bekommen

$$\frac{\begin{array}{lcl} \$\text{fac n} & = & \text{IF (= n 0) 1 (* n (\$fac (- n 1)))} \\ \$\text{Prog} & = & \$\text{fac 4} \end{array}}{\$\text{Prog}}$$

5.5.3 Generierung von Superkombinatoren mit graphischen Körpern

Bis jetzt hatte keiner unserer Superkombinatoren einen graphischen Körper. Dieser entsteht, wenn ein **letrec** freie Variablen enthält. Betrachten wir als Beispiel das Programm

$$\begin{array}{l} \textbf{letrec}\; Inf \quad = \quad \lambda\, v\, . \quad (\quad \textbf{letrec}\; vs \;=\; \text{CONS}\; v\; vs \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\; vs) \\ \qquad\qquad\quad \textbf{in}\; Inf\; 4 \end{array}$$

das die unendliche Liste aus Vieren berechnet. Wieder befindet sich *Inf* auf dem obersten Niveau und enthält keine innere Lambda-Abstraktion, ist also ein Superkombinator. Wir erhalten

$$\frac{\begin{array}{lcl} \$\text{Inf v} & = & \textbf{letrec}\ \text{vs} = \text{CONS v vs}\ \textbf{in}\ \text{vs} \\ \$\text{Prog} & = & \$\text{Inf 4} \end{array}}{\$\text{Prog}}$$

Der graphische Körper des Superkombinators bewahrt die (endliche) zyklische Repräsentation der (unendlichen) Datenstruktur.

Wir betrachten nun als Beispiel das Lambda-Liften eines *Miranda*-Programms, das die Summe der 100 ersten natürlichen Zahlen bestimmt.

```
sumInts m    = sum (count 1)
               where count n  =                    []    , n > m
                              = n : count (n + 1)
sum []       = 0
sum (n : ns) = n + sum ns

sumInts 100
```

Die Übersetzung in den erweiterten Lambda-Kalkül ergibt

```
letrec
    sumInts = λ n . letrec
                        count = λ n . IF (> n m) NIL
                                         (CONS n (count (+ n 1)))
                    in sum    (  count 1 )
    sum     = λ ns . IF       (= ns NIL) 0
                              (+ (HEAD ns) (sum (TAIL ns)))
in
sumInts 100
```

Die Variablen *sumInts* und *sum* sind auf dem obersten Niveau definiert, aber *sumInts* hat eine innere Lambda-Abstraktion. Diese $\lambda\, n$-Abstraktion hat die freien Variablen m und *count*. Wir liften sie nach außen, um einen Superkombinator zu erhalten, den wir $count nennen wollen.

```
$count count m n = IF (> n m) NIL
                      (CONS n (count (+ n 1)))
```

```
letrec
    sumInts = λ m . letrec
                      count = $count count m
                    in sum (count 1)
    sum     = λ ns . IF (= ns NIL)0
                        (+ (HEAD ns) (sum (TAIL ns)))
in sumInts 100
```

Nun haben *sumInts* sowie *sum* keine inneren Lambda-Abstraktionen mehr und sind auf dem obersten Niveau, so daß sie Superkombinatoren sind. Damit lautet das Ergebnis

```
$count count m n    =  IF (¿ n m) NIL
                           (CONS n (count (+ n 1)))
$sum ns             =  IF (= ns NIL) 0
                           (+ (HEAD ns) ($sum (TAIL ns)))
$sumInts m          =  letrec count = $count count m
                       in $sum (count 1)
$Prog               =  $sumInts 100
-------------------------------------------------------
$Prog
```

5.5.4 Eine andere Technik

Diese Technik ist nicht die einzige, um rekursive Funktionen zu Lambda-liften. Johnsson beschreibt in [Johnsson 1987] einen anderen Algorithmus, den wir hier kurz vorstellen.

Angenommen, wir haben ein Programm mit einer rekursiven Funktion f, die die freie Variable v enthält.

$$
\begin{array}{l}
(\ldots \\
\quad \textbf{letrec} \quad f \quad = \quad \lambda\, x\,.\,(\ldots f \ldots v \ldots) \\
\quad\quad \textbf{in} \quad\;\; (\ldots f \ldots) \\
\ldots)
\end{array}
$$

Wir erzeugen einen rekursiven Superkombinator \$f aus f durch Abstraktion der freien Variablen (in unserem Fall nur v), aber nicht von f selbst. Dafür ersetzen wir alle Verwendungen von \$f durch (\$f v), auch die im Körper von \$f selbst. Das ergibt

$$
\begin{array}{l}
\$\text{f v x} \quad = \quad \cdots\,(\$\text{f } v)\cdots v \cdots \\
\hline
(\ldots \\
\qquad\qquad\qquad (\cdots\,(\$\text{f } v)\,\cdots\,) \\
\ldots)
\end{array}
$$

Wir wollen diese Methode an unserem Beispiel illustrieren. Wir beginnen wieder mit dem Programm

```
letrec
    sumInts  =  λ n  .  letrec
                            count  =  λ n . IF (> n m) NIL
                                           (CONS n (count (+ n 1)))
                        in sum     (   count 1 )
    sum      =  λ ns .  IF          (= ns NIL) 0
                                    (+ (HEAD ns) (sum (TAIL ns)))
in
sumInts 100
```

Zuerst liften wir die $\lambda.n$-Abstraktion, wobei wir nur die freie Variable m (und nicht *count*) abstrahieren. Außerdem ersetzen wir alle Aufrufe von *count* durch ($count m); das ergibt

$$\$count\ m\ n \;=\; IF\ (>\ n\ m)\ NIL\ (CONS\ n\ (\$count\ m\ (+\ n\ 1)))$$

---

letrec

$$\begin{array}{lcl} sumInts & = & \lambda m\,.\,sum(\$count\ m\ 1) \\ sum & = & \lambda ns.IF(=\ ns\ NIL)\ 0 \\ & & (+\ (HEAD\ ns)\ (sum\ (TAIL\ ns))) \end{array}$$

in sumInts 100

Jetzt sind *sumInts* und *sum* Superkombinatoren, das Liften ergibt

$$\begin{array}{lcl} \$count\ m\ n & = & IF\ (>\ n\ m)\ NIL\ (CONS\ n\ (\$count\ m\ (+\ n\ 1))) \\ \$sum\ ns & = & IF\ (=\ ns\ NIL)\ 0 \\ & & (+\ (HEAD\ ns)\ (\$sum\ (TAIL\ ns))) \\ \$sumInts\ m & = & \$sum\ (\$count\ m\ 1) \\ \$Prog & = & \$sumInts\ 100 \end{array}$$

$Prog

Im Gegensatz zur vorigen Methode hat hier kein Superkombinator einen graphischen Körper; die gesamte Rekursion wird durch direkte Rekursion der Superkombinatoren behandelt.

Die neue Methode hat einen großen Vorteil. In der früheren Version wurde der rekursive Ruf von *count* im $count-Superkombinator zu einer Funktion gemacht, die als Argument (*count*) an $count übergeben wird. Die neue Methode macht den rekursiven Ruf direkt zum Superkombinator $count. Das heißt, daß der Compiler sieht, welche Funktion gerufen wird, und mit dieser Information kann er effizienteren Code generieren.

Auf der anderen Seite ist der durch die neue Methode erzeugte Code für den $count-Superkombinator größer, da er einen Anwendungs-Knoten ($count m) mehr enthält.

5.6 SK-Kombinatoren

In diesem Abschnitt wollen wir eine andere Graphenreduktions-Technik besprechen, die auf einer festen Menge von Superkombinatoren basiert. Die bedeutendsten Vertreter dieser Menge sind **S** und **K**, daher die Überschrift. Diese

Methode ist interessant, weil sie nur eine extrem einfache Reduktionsmaschine erfordert, die nur eingebaute Operatoren unterstützt und keinen Schablonen-Instanziierungs-Mechanismus braucht.

Das SK-Übersetzungsschema geht auf D. TURNER [TURNER 1979] zurück, wir wollen diesen Artikel aber hier nicht verwenden, damit wir keine neuen Schreibweisen einführen müssen. Wenn sich der Leser aber näher mit den SK-Kombinatoren beschäftigt, sollte er sich diesen Artikel unbedingt ansehen, zumal er extrem leicht verständlich ist.

5.6.1 Einführung in S, K und I

Unsere Strategie ist es, das Programm in ein anderes zu transformieren, das nur die eingebauten Funktionen und Konstanten sowie die Kombinatoren **S**, **K** und **I** enthält. Diese Kombinatoren sind definiert durch die folgenden Reduktionsregeln.

$$\begin{array}{lcl} \mathbf{S}\ \mathrm{f\ g\ x} & \longrightarrow & \mathrm{f\ x\ (g\ x)} \\ \mathbf{K}\ \mathrm{x\ y} & \longrightarrow & \mathrm{x} \\ \mathbf{I}\ \mathrm{x} & \longrightarrow & \mathrm{x} \end{array}$$

Warum wir gerade diese Kombinatoren verwenden, wird im folgenden klarer werden. **S, K** und **I** sind alles Superkombinatoren, da sie die Definition erfüllen. Wir verwenden aber den allgemeineren Begriff „Kombinator“, weil er von allen Autoren benutzt wird.

Als Beispiel sehen wir uns die Lambda-Abstraktion

$$\mathrm{Fun} \quad = \quad \lambda x\,.\,E_1\ \ E_2$$

an, in der E_1 und E_2 beliebige Ausdrücke sind. Durch Anwendung der S-Transformation, die sich aus der entsprechenden Reduktionsregel ergibt, können wir einen zu Fun äquivalenten Ausdruck finden.

$$\mathrm{Fun'} \quad = \quad \mathbf{S}\ \ (\lambda x\,.\,E_1)\ \ (\lambda x\,.\,E_2)$$

Wir zeigen, daß Fun und Fun' äquivalent sind, indem wir beide auf dasselbe Argument anwenden.

$$\begin{array}{rcl} \mathrm{Fun\ arg} = & (\lambda x\,.\,E_1\ \ E_2)\ \mathrm{arg} \\ \longrightarrow & (E_1[\mathrm{arg}/x])\ \ (E_2[\mathrm{arg}/x]) \end{array}$$

$$\begin{array}{rcl} \mathrm{Fun'\ arg} = & \mathbf{S}\ \ (\lambda x\,.\,E_1)\ \ (\lambda x\,.\,E_2)\ \ \mathrm{arg} \\ \longrightarrow & ((\lambda x\,.\,E_1)\ \mathrm{arg})\ \ ((\lambda x\,.\,E_2)\ \ \mathrm{arg}) \\ \longrightarrow & (E_1[\mathrm{arg}/x])\ \ (E_2[\mathrm{arg}/x]) \end{array}$$

Allgemein ist die S-Transformation durch folgende Regel beschrieben.

$$\lambda x\,.\,E_1\ \ E_2 \quad \Longrightarrow \quad \mathbf{S}\ \ (\lambda x\,.\,E_1)\ \ (\lambda x\,.\,E_2)$$

Wir sehen, daß wir bei jeder Anwendung der S-Transformation zwei neue Lambda-Abstraktionen, aber mit kleinerem Körper, erzeugen. Solange der Körper einer Lambda-Abstraktion eine Funktionsanwendung ist, führen wir eine S-Transformation aus. Zuletzt haben wir nur noch atomare Objekte als Körper, und hier müssen wir zwei Fälle betrachten.

1. Der Ausdruck ist $(\lambda\, x\, .\, x)$. Das ist gerade der Identitätskombinator **I**, und unter Verwendung der Reduktionsregel

$$\mathbf{I}\ x \quad \longrightarrow \quad x$$

   ersetzen wir $(\lambda\, x\, .\, x)$ durch **I**. Die I-Transformation lautet daher

$$\lambda\, x\, .\, x \quad \Longrightarrow \quad \mathbf{I}$$

2. Der Ausdruck ist $(\lambda\, x\, .\, c)$, wobei c eine Konstante oder eine von x verschiedene Variable ist. Dies ist eine Funktion, die ihr Argument nicht beachtet und immer c als Wert liefert. Daher können wir sie durch $(\mathbf{K}\ c)$ ersetzen, da

$$\mathbf{K}\ c\ x \quad \longrightarrow \quad c$$

   Die K-Transformation ist also

$$\lambda\, x\, .\, c \quad \Longrightarrow \quad \mathbf{K}\ c$$

   wobei c eine Konstante oder eine von x verschiedene Variable ist.

5.6.2 Compilation

Die **S**-, **K**- und **I**-Transformationen bilden einen vollständigen Compilations-Algorithmus, der jeden Lambda-Ausdruck E in eine Form bringt, die nur noch **S**, **K**, **I** und Konstantan enthält.

SOLANGE E eine Lambda-Abstraktion enthält **MACHE**

1. Wähle eine innerste Lambda-Abstraktion aus E.
2. Ist ihr Körper eine Anwendung, führe eine S-Transformation aus.
3. Sonst wende, je nach dem Aussehen des Lambda-Körpers, die K- oder die I-Transformation an.

ENDE

Indem wir die innerste Lambda-Abstraktion zuerst transformieren, sichern wir, daß der Körper der Ausgewählten keine Lambdas enthält. Daher treten keine Namenskonflikt-Probleme auf.

Am Ende erhalten wir einen Ausdruck, der keine Variablen mehr enthält. Wir sehen am nachstehenden Beispiel, daß wir auch Unterausdrücke mit freien Variablen transformieren können.

$$\begin{array}{rcl} \lambda x\,.\,\lambda y\,.\,x & \Longrightarrow_K & \lambda x\,.\,\mathbf{K}\ x \\ & \Longrightarrow_S & \mathbf{S}\ (\lambda x\,.\,\mathbf{K})\ (\lambda x\,.\,x) \\ & \Longrightarrow_K & \mathbf{S}\ (\mathbf{K}\ \mathbf{K})\ (\lambda x\,.\,x) \\ & \Longrightarrow_I & \mathbf{S}\ (\mathbf{K}\ \mathbf{K})\ \mathbf{I} \end{array}$$

Als Index am Pfeil steht, welche Transformation wir ausgeführt haben. Wir prüfen unser Beispiel, indem wir Ausgangs- und Endausdruck auf zwei feste Argumente, z. B. 5 und 6, anwenden.

$$\begin{array}{rcl} ((\lambda x\,.\,\lambda y\,.\,x)\ 5)\ 6 & \longrightarrow & (\lambda y\,.\,5)\ 6 \\ & \longrightarrow & 5 \end{array}$$

$$\begin{array}{rcl} (\mathbf{S}\ (\mathbf{K}\ \mathbf{K})\ \mathbf{I})\ 5)\ 6 & \longrightarrow & ((\mathbf{K}\ \mathbf{K})\ \mathbf{5}\ (\mathbf{I}\ 5))\ 6 \\ & \longrightarrow & ((\mathbf{K}\ \mathbf{K})\ 5\ 5)\ 6 \\ & \longrightarrow & (\mathbf{K}\ 5)\ 6 \\ & \longrightarrow & 5 \end{array}$$

Beide Male kommen wir zum selben Ergebnis. Unser Beispiel können wir noch optimieren, da sich die folgende Optimierungsregel anwenden läßt.

$$\mathbf{S}\ (\mathbf{K}\ p)\ \mathbf{I} \quad = \quad p$$

Diese können wir wieder durch Anwendung beider Seiten auf ein beliebiges Argument x beweisen.

$$\begin{array}{rcl} \mathbf{S}\ (\mathbf{K}\ p)\ \mathbf{I}\ x & \longrightarrow & (\mathbf{K}\ p)\ x\ (\mathbf{I}\ x) \\ & \longrightarrow & (\mathbf{K}\ p)\ x\ x \\ & \longrightarrow & p\ x \end{array}$$

Daß das auch auf der rechten Seite herauskommt, brauchen wir wohl nicht extra aufzuschreiben.

Wir erhalten schließlich folgende Transformation.

$$\lambda x\,.\,\lambda y\,.\,x \quad \Longrightarrow \quad \mathbf{K}$$

Wir haben richtig gerechnet, denn das ist unsere Definition des **K**-Kombinators.

5.6.3 Implementation

Die Kombinatoren **S, K** und **I** sind Superkombinatoren, daher ist die Reduktionsmaschine, die zu ihrer Abarbeitung benötigt wird, eine abgerüstete Version der Superkombinator-Reduktionsmaschine. Die Methode des Findens des

nächsten reduzierbaren Ausdrucks durch Absteigen im Rückgrat, die Verwendung von Verweis-Knoten und die Implementation der Rekursion mit dem **Y**-Kombinator funktionieren wie gehabt. Die Hauptunterschiede sind folgende.

- Die Kombinatoren sind in der Reduktionsmaschine direkt als eingebaute Funktionen implementiert, und nicht indirekt durch einen allgemeinen Superkombinator-Instanziierungsmechanismus.
- Die Reduktionsmaschine braucht keinen Schablonen-Instanziierungsmechanismus, da es keine Lambda-Abstraktionen zum Instanziieren gibt.

Ein auf der SK-Reduktion basierender Graphenreduzierer ist also eine der einfachsten Implementationen der Graphenreduktion.

5.6.4 I ist nicht nötig

Interessant ist, daß **S** und **K** auch ausreichen, da der Ausdruck (**S K K**) wertmäßig gleich **I** ist. Das ist jedoch nur von theoretischem Interesse, alle vernünftigen Implementationen enthalten **I**.

$$\begin{aligned} \mathbf{S}\ \mathbf{K}\ \mathbf{K}\ x &\longrightarrow \mathbf{K}\ x\ (\mathbf{K}\ x) \\ &\longrightarrow x \\ \\ \mathbf{I}\ x &\longrightarrow x \end{aligned}$$

Daher haben wir diesen Abschnitt auch „SK-Kombinatoren“, und nicht „SKI-Kombinatoren“ genannt.

5.6.5 Vergleich mit Superkombinatoren

SK-Kombinatoren repräsentieren ein Extrem von Graphenreduktionstechniken. Komplexe Reduktionen werden zu einer Komposition von vielen schnellen und einfachen reduziert, so daß das „Korn“ der Ausführung so klein wird, wie man nur denken kann.

Für die SK-Kombinatoren sprechen

- Eine kleine, feste Menge von Kombinatoren kann direkt in Hardware implementiert werden, damit umgehen wir ein Niveau der Interpretation.
- Die Technik ist vollständig träge.
- Die Reduktionsmaschine ist relativ einfach zu implementieren.

Gegen die SK-Kombinatoren sprechen folgende Tatsachen.

- Das „Korn“ der Abarbeitungsschritte ist zu klein. Da die Argumente einer Funktion jeweils um ein Niveau im Körper absteigen, werden viele Zwischen-Anwendungs-Knoten erzeugt und fast sofort wieder auseinandergenommen. Das heißt, daß ein SK-Kombinator-Reduzierer viel Zwischenspeicher benötigt und damit dem *Garbage collector* eine größere Last aufbürdet.
- Die Übersetzung in Kombinatoren ist aufwendiger gegenüber den Superkombinator-Techniken, und das resultierende Programm ist größer.

Kapitel 6

Die G-Maschine

In diesem Kapitel sehen wir uns die G-Maschine an, eine extrem schnelle Implementation der Graphenreduktion, die auf Superkombinator-Compilation basiert. Die G-Maschine wurde von T. Johnsson und L. Augustsson [Johnsson 1987, Augustsson 1987] entwickelt.

Die *G-Maschine* ist eine Maschine mit den folgenden Komponenten:

- **S**, dem Stack
- **G**, dem Graphen
- **C**, der Codefolge, die noch auszuführen ist
- **D**, dem Dump, der ein Stack aus Paaren (S, C) ist, wobei S ein Stack und C eine Codefolge ist.

Ein G-Maschinenzustand ist durch das Quadrupel $\langle S, G, C, D\rangle$ vollständig beschrieben. Die Operationen der G-Maschine kann man dann, wie wir es bei der SECD-Maschine getan haben, in Form von Zustandsübergängen angeben; wir wollen sie in dieser Übersicht jedoch nur verbal einführen.

Wenn wir Superkombinator-Körper in eine Folge von Instruktionen übersetzen wollen, benötigen wir eine Sprache, in der diese Instruktionen geschrieben werden. Die Maschinensprache eines bestimmten Rechners zu verwenden wäre schlecht, weil wir erstens immer von vorn anfangen, wenn wir einen Codegenerator für eine andere Maschine haben wollen, und uns zweitens der Gefahr aussetzen könnten, das Problem der Superkombinator-Compilation mit dem Problem, die Möglichkeiten des verwendeten Rechners auszureizen, zu vermischen. Wir verwenden daher einen Zwischencode, den G-Code, in den die Superkombinatoren übersetzt werden.

Der Compiler für die G-Maschine arbeitet nach folgendem Muster.

- Frühe Phasen führen Typprüfung, Übersetzung der Mustererkennung usw. aus und transformieren das Quellprogramm in den durch **let** und **letrec** erweiterten Lambda-Kalkül.

- Ein Lambda-Lifter überführt das Programm in Superkombinator-Form.

- Die Superkombinatoren werden in G-Code übersetzt.

- Zuletzt wird aus dem G-Code der Maschinencode des verwendeten Rechners erzeugt.

Im folgenden geben wir einen Überblick über die Transformation der Superkombinator-Definitionen in G-Code und beschreiben dabei kurz die wichtigsten Instruktionen der G-Maschine.

6.1 Die Quellsprache des G-Compilers

Die Compilation in G-Code beginnt mit einem Programm, das aus einer Menge von Superkombinator-Definitionen der Form

$$\$S \;\; x_1 \;\; x_2 \;\; \ldots \;\; x_n \quad = \quad E$$

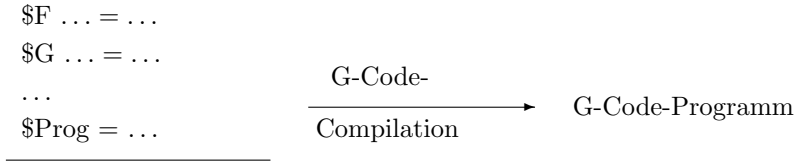
besteht, wobei E ein Ausdruck ohne Lambdas ist, der aber **let**'s und **letrec**'s enthalten darf. Die lokalen Funktionsdefinitionen hat der Lambda-Lifter entfernt. Der Ausdruck E ist also wie folgt aufgebaut.

```
<E>  ::=  <Konstante>
        |  <Variable>
        |  <E> <E>
        |  let <Variable> = <E> in <E>
        |  letrec <Variable> = <E>
                  ...
                  <Variable> = <E>
           in <E>
```

Wir haben die Syntax formal angegeben, da unser Compiler für jedes Konstrukt einen extra Fall benötigt.

6.2 Compilation in G-Code

Der G-Code-Compilations-Algorithmus benimmt sich wie folgt.



Er nimmt eine Menge von Superkombinator-Definitionen sowie $Prog und produziert daraus ein G-Code-Programm. Dieses Programm besteht aus den folgenden Teilen:

1. Einem Segment aus Initialisierungs-Code, der die notwendige Laufzeit-Initialisierung ausführt.
2. Einem Segment von G-Code, der den Superkombinator $Prog auswertet und seinen Wert druckt. Dieses Segment folgt gleich nach (1).
3. Einem Segment, das den G-Code aller Superkombinatoren enthält. Der Code eines jeden Superkombinators wird durch eine bestimmte Marke identifiziert.
4. Markierten Segmenten von G-Code für jede eingebaute Funktion. Diese stellen die Laufzeit-Bibliothek dar, da sie bei allen Programmen gleich sind.

Die Code-Segmente für (1) und (2) sind relativ einfach. Für (1) brauchen wir einen G-Code-Befehl BEGIN, der den Programmbeginn markiert und die Initialisierungen ausführt. Zur Auswertung von $Prog bringen wir es zuerst auf den Stack (mit dem Befehl PUSHGLOBAL), werten es mit dem EVAL-Befehl aus und drucken das Ergebnis mit PRINT. Hier ist eine Code-Folge, die das System initialisiert und den Wert von $Prog druckt.

BEGIN;	Beginn des Programms
PUSHGLOBAL $Prog;	Bringt $Prog auf den Stack
EVAL;	Auswertung
PRINT;	Druck des Ergebnisses
END;	Programmende

Die G-Maschinen-Implementation ist in fünf Übersetzungs-Schemata unterteilt.

- Das **C-Schema**, das den Code für die Konstruktion von Graphen, die Ausdrücke repräsentieren, generiert.

- Das **E-Schema**, das den Code zur Auswertung von Ausdrücken bis zur schwachen Kopf-Normalform erzeugt. Ein Ausdruck ist durch einen Graphen repräsentiert, der durch den vom C-Schema erzeugten Code konstruiert wird. Nach der Auswertung wird ein Zeiger auf das Ergebnis auf dem Stack abgelegt.

- Das **F-Schema** liefert aus einer Superkombinator-Definition den compilierten G-Code.

- Das **R-Schema** compiliert den Code für den Körper eines Superkombinators.

- Das **B-Schema**, das ähnlich zum C-Schema ist, compiliert Code zur Auswertung von Integer- und Boolean-wertigen Ausdrücken, wobei das Resultat auf einem anderen Stack, dem Dump, abgelegt wird. Dieses Schema trennt die Behandlung der primitiven Funktionen ab, die strikte Semantik erfordern, und erhöht die Effizienz der grundlegenden arithmetischen, logischen und Bedingungsoperationen, die ihre Argumente von der Spitze des Dump beziehen, so daß der Aufbau von Graphen gespart wird.

Der Dump enthält atomare Datenwerte und die bei der Definition der G-Maschine aufgeführten Zustandsinformationen, die bei der Aktivierung von Auswertungen gerettet werden, während auf dem Stack nur Zeiger in den Graphen des gerade reduzierten Ausdrucks abgelegt sind.

Wir wenden uns nun dem letzten offenen Problem, der Compilation von Superkombinatoren, zu.

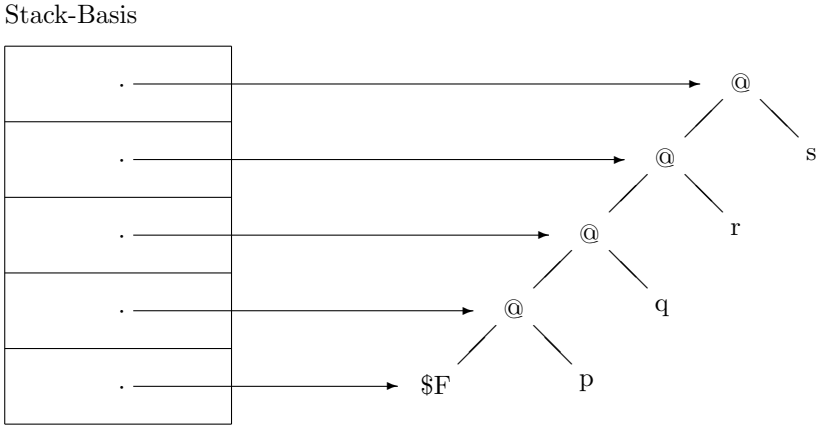
6.3 Übersetzung einer Superkombinator-Definition

Wie gesagt, erzeugt das F-Schema aus einer Superkombinator-Definition den Code für den Funktionskörper.

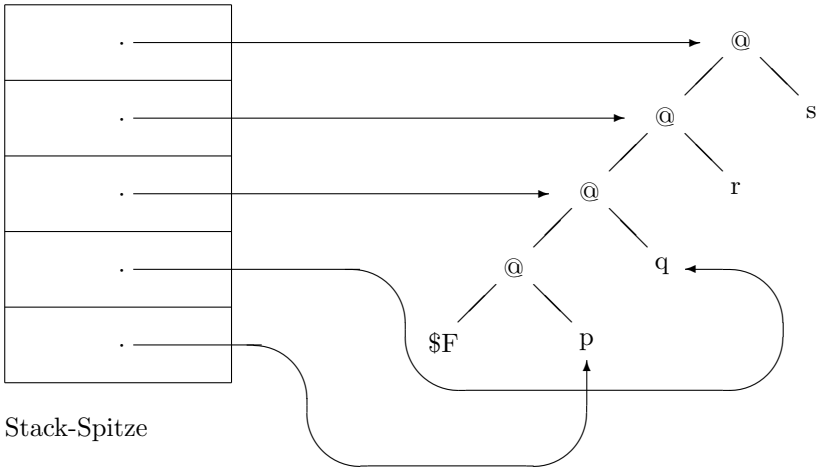
$$\mathbf{F}[\$F\ x_1\ \ldots x_n\ =\ E]\quad =\quad \ldots \text{ G-Code für } \$F\ \ldots$$

6.3.1 Stacks und Kontexte

Angenommen, die G-Maschine soll den Ausdruck (\$F p q r s), wobei \$F ein Superkombinator von zwei Argumenten ist, auswerten. Nachdem das Rückgrat des Graphen entfaltet ist, sieht der Stack so aus.

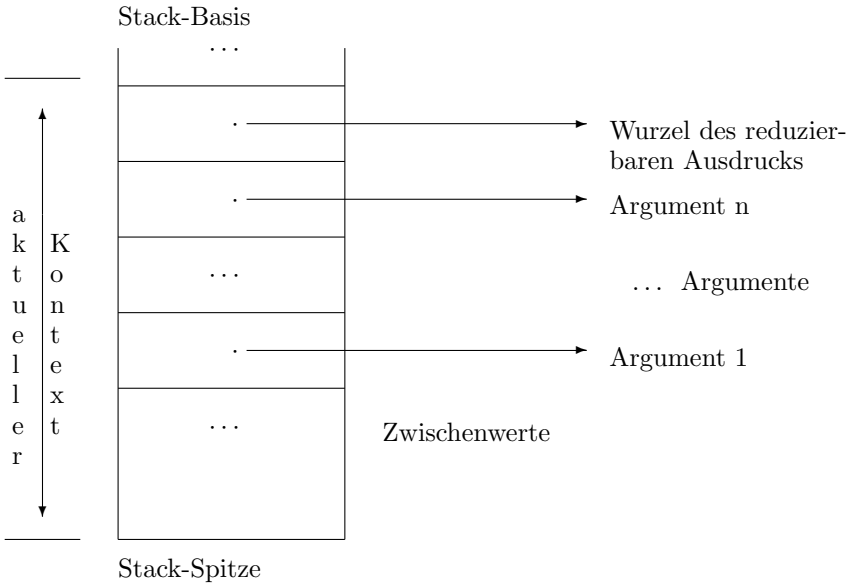


(In allen Abbildungen wächst der Stack nach unten.) Das ist nicht die bequemste Konfiguration während der Ausführung von $F, weil der Zugriff auf die Argumente p und q indirekt über die Wirbel erfolgen muß. Wir ordnen daher, nachdem wir mit dem Entfalten des Rückgrates fertig sind und bevor wir mit der Reduktion beginnen, den Stack so um, daß die Elemente des Stacks direkt auf die Argumente zeigen.



Weiterhin benötigen wir einen Zeiger auf die Wurzel des reduzierbaren Aus-

drucks, weil wir diese nach der Reduktion aktualisieren wollen. Jetzt sind p und q bequem erreichbar. Der Superkombinator $F wurde vom Stack entfernt, da die Stack-Umordnung in Wirklichkeit von einem Vorspann des Zielcodes für $F ausgeführt wird. Während der Abarbeitung von $F hat der Stack das nachstehend abgebildete Aussehen.



Der Abschnitt von der Stack-Spitze bis einschließlich zum Zeiger auf die Wurzel des reduzierbaren Ausdrucks ist der **aktuelle Kontext**. Er sitzt immer an der Spitze des Stacks, aber es können sich weitere Stack-Elemente zwischen der Stack-Basis und der Basis des aktuellen Kontextes befinden. Am Ende der Ausführung einer Funktion werden die Wurzel des reduzierten Ausdrucks aktualisiert und alle Einträge des aktuellen Kontextes entfernt, so daß nur noch der Zeiger auf die Wurzel des Ergebnisses auf dem Stack verbleibt.

Daraus können wir zwei Grundregeln ersehen.

- Wenn die Abarbeitung des Codes, der einem Superkombinator entspricht, beginnt, sind die Argumente an der Stack-Spitze, und darunter steht ein Zeiger auf die Wurzel des zu reduzierenden Ausdrucks.
- Ist die Ausführung eines Superkombinators beendet, befindet sich nur der Zeiger auf den reduzierten Graphen auf dem Stack. Der reduzierte Graph ist nicht unbedingt in schwacher Kopf-Normalform, und so initiiert der letzte Befehl im Superkombinator die nächste Reduktion.

Während der Compilation eines Superkombinators hält der Compiler ein Modell des Stacks. Insbesondere muß er wissen, wo der Wert jeder Variablen — relativ

zur Stack-Spitze — steht. Für alle unsere Compilations-Funktionen wird diese Information gehalten als

- p, eine Funktion, die einen Bezeichner als Argument hat und eine Zahl liefert, die die Verschiebung des Wertes dieses Bezeichners relativ zur Basis des aktuellen Kontextes darstellt. Das Basiselement des aktuellen Kontextes (der Zeiger auf die Wurzel des reduzierbaren Ausdrucks) hat die Verschiebung 0, das letzte Argument die Verschiebung 1 usw.
- d, die Tiefe des aktuellen Kontextes minus 1.

Aus diesen können wir die Verschiebung einer Variablen x gegenüber der Stack-Spitze als $(d - p\ x)$ berechnen, wobei das Element an der Spitze die Verschiebung 0 hat.

6.3.2 Das R-Schema

Wir können nun die komplette Definition des F-Schemas geben.

$$\begin{array}{lcl} \mathbf{F}[f\ x_1\ \ldots\ x_n\ =\ E] & = & \text{GLOBSTART}\ f,\ n; \\ & & \mathbf{R}[E]\ [x_1 = n,\ x_2 = n-1,\ \ldots,\ x_n = 1]\ n \end{array}$$

wobei f der Name des Superkombinators ist. Der Befehl "GLOBSTART f, n" markiert den Beginn der Funktion f, die n Argumente hat. Danach ruft **F** eine Funktion **R**, die den Körper E compiliert, und übergibt ihr die Werte für p und d in dieser Reihenfolge.

Wir beschreiben nun, was **R** tut. Wie wir an unserem Beispiel gesehen haben, hat der Code eines Superkombinators vier Dinge zu erledigen.

1. Konstruktion einer Instanz des Superkombinator-Körpers unter Verwendung der Parameter auf dem Stack
2. Aktualisieren der Wurzel des zu reduzierenden Ausdrucks mit einer Kopie der Wurzel des Ergebnisses
3. Entfernen der Parameter vom Stack
4. Initiieren der nächsten Reduktion.

Dies führt direkt zum Übersetzungsschema für **R**.

$$\begin{array}{lcl} \mathbf{R}[E]\ p\ d & = & \mathbf{C}[E]\ p\ d; \\ & & \text{UPDATE}\ d+1; \\ & & \text{POP}\ d; \\ & & \text{UNWIND} \end{array}$$

Wir benutzen eine andere Hilfsfunktion, nämlich **C** (für „Konstruiere Instanz“), die den Code zur Konstruktion einer Instanz von E, wobei ein Zeiger auf die Instanz auf dem Stack abgelegt wird, produziert. Das bildet Schritt (1). Der UPDATE-Befehl überschreibt die Wurzel des reduzierten Ausdrucks (der nun die Verschiebung $(d+1)$ gegenüber der Stack-Spitze hat) mit der neu erzeugten Instanz, die gerade an der Stack-Spitze ist (Schritt (2)) und entfernt den Zeiger auf die Instanz vom Stack. Der POP-Befehl entfernt die Argumente vom Stack (Schritt (3)), und UNWIND veranlaßt die nächste Reduktion (Schritt (4)).

6.3.3 Das C-Schema

Das C-Schema produziert Code zur Konstruktion einer Instanz eines Ausdrucks. Es ist eine Funktion mit dem folgenden Verhalten.

- **Argumente**: Der zu compilierende Ausdruck sowie p und d, die angeben, wo auf dem Stack sich die Argumente des Superkombinators befinden.
- **Ergebnis**: Eine G-Code-Folge, die bei der Ausführung eine Instanz des Ausdrucks konstruiert, wobei die Vorkommen der formalen Parameter durch Zeiger auf die entsprechenden Superkombinator-Argumente ersetzt sind, und einen Zeiger auf diese Instanz auf dem Stack hinterläßt.

Um **C** zu definieren, müssen wir das Ergebnis des Aufrufs

$$\mathbf{C}[E]\ p\ d$$

für jede Form des Ausdruckes E angeben.

E ist eine Konstante

Hier gibt es zwei Fälle zu betrachten. Erstens kann E eine ganze Zahl i (oder ein anderer eingebauter Konstanten-Wert) sein. Dann brauchen wir nur einen Zeiger auf die Zahl auf den Stack zu befördern. Wir erhalten damit

$$\mathbf{C}[i]\ p\ d = \text{PUSHINT}\ \ i$$

Zweitens kann E ein Superkombinator oder eine eingebaute Funktion f sein. Wir müssen einen Zeiger auf die Fuktion auf den Stack bringen, und bekommen

$$\mathbf{C}[f]\ p\ d = \text{PUSHGLOBAL}\ \ f$$

E ist eine Variable

E ist ein Variable x. Ihr Wert befindet sich im Stack mit der Verschiebung $(d - px)$ gegenüber der Spitze; und wir kopieren ihn an die Stack-Spitze.

$$\mathbf{C}[x]\ p\ d = \text{PUSH}\ \ (d\ -\ p\ x)$$

E ist eine Anwendung

Wenn E eine Anwendung $(E_1\ \ E_2)$ mit beliebigen Ausdrücken E_1 und E_2 ist, ist der zu konstruierende Ausdruck die Anwendung von E_1 auf E_2. Wir konstruieren daher zuerst eine Instanz von E_2 (danach befindet sich ein Zeiger auf diese Instanz an der Spitze des Stack) und dann eine von E_1. Nun erzeugen wir eine Anwendungs-Zelle aus den beiden Objekten an der Stack-Spitze und hinterlassen einen Zeiger auf den erzeugten Anwendungs-Knoten an der Stack-Spitze. Das wird erreicht durch die folgende Regel.

$$\begin{array}{lcl} \mathbf{C}[E_1\ \ E_2]\ p\ d & = & \mathbf{C}[E_2]\ p\ d; \\ & & \mathbf{C}[E_1]\ p\ (d+1); \\ & & \text{MKAP} \end{array}$$

Wir beachten, daß der Kontext während der Ausführung des zweiten **C**-Rufs um 1 größer ist; daher übergeben wir (d+1) anstatt d.

MKAP ist eine Instruktion, die die beiden Objekte an der Spitze des Stack nimmt, sie vom Stack entfernt, einen @-Knoten auf dem Heap formt und einen Zeiger auf diesen Knoten auf den Stack bringt.

E ist ein let-Ausdruck

Als nächstes behandeln wir die Regel für **let**-Ausdrücke

$$\mathbf{C}[\mathbf{let}\ x\ =\ E_x\ \mathbf{in}\ E_b]\ p\ d$$

wobei x eine Variable ist und E_x sowie E_b Ausdrücke sind. Ein **let** in einem Superkombinator-Körper ist ein Weg der Beschreibung eines Graphen (mit Teilung) anstatt eines Baumes. Wir können ein **let** in der folgenden Weise übersetzen.

- Zuerst konstruieren wir eine Instanz von E_x und bringen einen Zeiger darauf auf den Stack.
- Dann erweitern wir p so, daß x an der Stelle $(d+1)$ von der Basis des Kontextes aus gefunden werden kann (das stimmt, weil sich der Wert von x (die erzeugte Instanz) an der Stack-Spitze befindet).
- Dann konstruieren wir, unter Verwendung der neuen Werte für p und d, eine Instanz von E_b und bringen einen Zeiger auf diese Instanz auf den Stack.
- Jetzt befindet sich ein Zeiger auf die Instanz von E_b an der Spitze des Stacks, und darunter ist ein Zeiger auf die Instanz von E_x. Wir benötigen den letzteren nicht länger, und wir streichen ihn aus dem Stack, indem wir das Element an der Spitze um eine Position zur Basis hin gleiten lassen.

$$\begin{aligned} & \mathbf{C}[\mathbf{let}\ x\ =\ E_x\ \mathbf{in}\ E_b]\ p\ d \\ = \quad & \mathbf{C}[Ex]\ p\ d; \\ & \mathbf{C}[E_b]\ p[x = d + 1]\ (d + 1); \\ & \text{SLIDE 1} \end{aligned}$$

Der Befehl "SLIDE 1" streicht ein Element aus dem Stack.

E ist ein letrec-Ausdruck

Zuletzt betrachten wir die Regel für

$$\mathbf{C}[\mathbf{letrec}\ D\ \mathbf{in}\ Eb]\ p\ d$$

wobei D eine Menge von Definitionen und Eb ein Ausdruck ist. Ein **letrec** in einem Superkombinator-Körper ist eine Beschreibung eines zyklischen Graphen. Der Weg zur Konstruktion eines solchen Graphen ist der folgende.

1. Zuerst fordern wir leere Zellen, eine für jede Definition, an und bringen Zeiger darauf auf den Stack. Diese leeren Zellen werden Löcher genannt.

2. Nun erweitern wir den durch p und d beschriebenen Kontext, um zu sagen, daß die im **letrec** gebundenen Variablen an den eben zugeordneten Stack-Plätzen gefunden werden können.

3. Für jeden Definitions-Körper

   (a) konstruieren wir eine Instanz davon und bringen einen Zeiger auf diese Instanz auf den Stack; dann

   (b) aktualisieren wir das entsprechende Loch mit der Instanz, wozu wir den UPDATE-Befehl verwenden, der auch die Zeiger an der Stack-Spitze entfernt.

   Während des Instanziierungs-Prozesses werden die Vorkommen der im **letrec** gebunden Variablen durch Zeiger auf das entsprechende Loch ersetzt, da wir den Kontext im Stadium (2) erweitert hatten.

4. Nun instanziieren wir Eb und hinterlassen einen Zeiger auf die Instanz an der Stack-Spitze.

5. Zuletzt streichen wir die Zeiger auf die Definitions-Körper aus dem Stack. Das ist möglich, weil der SLIDE-Befehl ein Argument hat, das angibt, wieviele Elemente zu streichen sind.

Wir erhalten

$$\mathbf{C}[\textbf{letrec } D \textbf{ in } Eb]\ p\ d$$

$$\begin{array}{rl} = & \textbf{CLetrec}\ [D]\ p'\ d'; \\ & \mathbf{C}[Eb]\ p'\ d'; \\ & \text{SLIDE}\ (d' - d) \end{array}$$

$$\textbf{wobei}(p',\ d')\ =\ \mathbf{Xr}[D]\ p\ d$$

Diese Form verwendet die beiden Hilfsfunktionen **CLetrec** und **Xr**, die wie folgt definiert sind.

$$\begin{array}{l} \textbf{CLetrec}[x_1 = E_1,\ x_2 = E_2,\ \ldots,\ x_n = E_n]\ p\ d \\ \quad = \quad \text{ALLOC}\ n; \\ \qquad \mathbf{C}[E_1]\ p\ d;\ \text{UPDATE}\ n; \\ \qquad \mathbf{C}[E_2]\ p\ d;\ \text{UPDATE}\ n-1; \\ \qquad \ldots \\ \qquad \mathbf{C}[E_n]\ p\ d;\ \text{UPDATE}\ 1 \end{array}$$

CLetrec führt die ersten beiden Schritte des Übersetzungsprozesses aus.

$$\begin{array}{l} \mathbf{Xr}[x_1 = E_1,\ x_2 = E_2,\ \ldots,\ x_n = E_n]\ p\ d \\ \quad = \quad (\quad p[x_1 = d+1,\ x_2 = d+2,\ \ldots,\ x_n = d+n], \\ \qquad d+n) \end{array}$$

Xr berechnet das erweiterte p sowie den neuen Wert von d und gibt beide als Paar zurück.

Es tritt ein Problem auf, wenn ein Definitions-Körper nur aus einer Variablen besteht, die im gleichen **letrec** gebunden wird; z. B. bei

$$\begin{array}{c} \textbf{letrec} \\ \\ \begin{array}{rcl} x & = & y \\ y & = & \text{CONS}\ 1\ y \end{array} \\ \textbf{in}\ E \end{array}$$

Hier versucht nämlich der UPDATE-Befehl, ein Loch mit einem anderen zu aktualisieren. Die unnötige Definition $x = y$ kann jedoch beim Lambda-Liften entfernt werden.

6.4 Superkombinatoren mit null Argumenten

Der Lambda-Lifting-Algorithmus kann Superkombinatoren ohne Argumente erzeugen. Ein offensichtliches Beispiel dafür ist der $Prog-Superkombinator. Solche Superkombinatoren sind einfach Konstanten-Ausdrücke, weil sie überhaupt keine Parameter haben. Wie sollen wir nun solche Ausdrücke compilieren?

- Wir compilieren sie überhaupt nicht, sondern halten sie als Teile des Graphen. Da sie keine Funktionen sind, werden sie niemals kopiert, so daß sie ohne weiteres aufgeteilt werden können. Dies ist eine akzeptable Lösung, aber es bedeutet, daß das compilierte Programm eine Mixtur von Maschinencode und Graph ist.

- Wir behandeln sie als Superkombinatoren mit null Argumenten und compilieren sie zu G-Code, der bei der Ausführung eine Instanz ihres Graphen konstruiert. Der Vorteil davon ist, daß das übersetzte Programm fast vollständig aus Maschinencode besteht.

6.5 Die eingebauten Funktionen

Bei den eingebauten Funktionen unterscheiden wir solche, die ihre Argumente auswerten, von denen, die sie nicht auswerten. Eine von denen, die ihr Argument auswerten, ist $NEG, zu der das folgende Code-Stück gehört.

EVAL;	Werte das Argument (das sich an der Stack-Spitze befindet) aus.
NEG;	Negiere es.
UPDATE 1;	Aktualisiere die Wurzel (der Zeiger darauf hat im Stack die Verschiebung 1).
UNWIND	Setze mit der Reduktion fort.

Der EVAL-Befehl tut das folgende.

- Er sieht sich das Objekt an der Spitze des Stacks an. Ist es eine CONS-Zelle, eine Zahl (ein Wahrheitswert bzw. ein Zeichen), ein Superkombinator oder eine eingebaute Funktion, macht EVAL überhaupt nichts.

- Ist es eine Anwendungs-Zelle, erzeugt EVAL einen neuen Stack, legt dort das Objekt von der Spitze des alten Stacks ab, sichert den Programmzähler (der auf den Befehl nach EVAL zeigt) und führt einen UNWIND-Befehl aus.

Nach jeder Reduktion wird ein UNWIND ausgeführt. Erkennt dieses UNWIND, daß der Ausdruck bereits in schwacher Kopf-Normalform ist, schreibt er den alten Stack zurück und springt zu der gesicherten Rückkehradresse.

Wir können den neuen Stack direkt auf der Spitze des alten aufbauen, beide können sich sogar um einen Eintrag überlappen, da das Spitzen-Element des alten dasselbe wie das Basis-Element des neuen Stacks ist. Wir sichern zwei Dinge auf einen anderen Stack, den Dump:

- den alten Stack-Zeiger und

- den alten Programmzähler.

Der UNWIND-Befehl am Ende des $NEG-Codes erkennt immer, daß die Auswertung zuende ist, weil das Ergebnis wieder eine Zahl ist. Es ist also verschwenderisch, UNWIND testen zu lassen, ob sich das Ergebnis in schwacher Kopf-Normalform befindet. Wir verwenden daher die Instruktion RETURN, die annimmt, daß sich der ausgewertete Ausdruck in schwacher Kopf-Normalform befindet, sich aber sonst wie UNWIND verhält.

Funktionen, die ihre Argumente nicht auswerten, verwenden den Befehl EVAL nicht.

Zur Erzeugung des Codes für $IF benötigen wir die Befehle JUMP, JFALSE sowie die Pseudo-Instruktion LABEL. Der Code für $IF ist.

```
PUSH    0;      Stapele erstes Argument
EVAL;           Werte es aus
JFALSE  L1;     Springe zu L1, wenn FALSCH
PUSH    1;      Stapele zweites Argument (bei WAHR)
JUMP    L2;

LABEL   L1;
PUSH    2;      Stapele drittes Argument (bei FALSCH)

LABEL   L2;
EVAL;
UPDATE  4;      Überschreibe die Wurzel
UNWIND
```

Wir dürfen am Ende nicht RETURN verwenden, obwohl das Ergebnis in schwacher Kopf-Normalform ist. Betrachten wir den Ausdruck

$$(\$\text{IF}\ \ E\ \ E_1\ \ E_2)\ \ 3$$

wobei E, E_1 und E_2 Ausdrücke sind. Hier wertet $IF das erste Argument und, in Abhängigkeit vom Resultat, entweder das zweite oder das dritte Argument aus, überschreibt den reduzierten Ausdruck ($IF $E\ E_1\ E_2$) und wendet das Ergebnis auf 3 an. Die Auswertung des Gesamtausdrucks ist noch nicht zuende, bloß weil sich das Ergebnis von ($IF $E\ E_1\ E_2$) in schwacher Kopf-Normalform befindet.

Damit haben wir einen vereinfachten Algorithmus, um unsere funktionalen Programme in G-Code zu überführen. Der Codegenerator für die Zielmaschine ersetzt dann die G-Code-Befehle durch Folgen von Maschineninstruktionen.

Literaturverzeichnis

Aho 1986 Aho, A.V., R. Sethi, J.D. Ullman: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, Massachusetts 1986.

Appel 1987 Appel, A.W., D.B. MacQueen: A Standard ML Compiler. Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture, Portland. Lecture Notes in Computer Science Vol. 274, Springer-Verlag, Berlin 1987, pp. 301 – 324.

Augustsson 1987 Augustsson, L.: Compiling Lazy Functional Languages, Part II. Ph.D. Thesis. Chalmers University of Technology, Department of Computer Sciences, Göteborg 1987.

Augustsson 1989 Augustsson, L., T. Johnsson: The Chalmers Lazy-ML Compiler. Computer Journal Vol. 32 (1989) No. 2, pp. 127 – 141.

Backus 1978 Backus, J.: Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Communications of the ACM Vol. 21 (1978) No. 8, pp. 613 – 641.

Burstall 1980 Burstall, R.M., D.B. MacQueen, D.T. Sanella: HOPE: An Experimental Applicative Language. Proceedings of the 1980 LISP Conference. Stanford, California 1980, pp. 136 – 143.

Cardelli 1984 Cardelli, L.: Compiling a Functional Language. Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. Austin, Texas 1984, pp. 208 – 217.

Cardelli 1985 Cardelli, L., P. Wegner: On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys Vol. 17 (1985) No. 4, pp. 471 – 522.

Clarke 1980 Clarke, T.J.W., P.J.S. Gladstone, C. Maclean, A.C. Norman: SKIM - The S, K, I Reduction Machine. Proceedings of the 1980 LISP Conference. Stanford, California 1980, pp. 128 – 135.

FAIRBRAIN 1987 Fairbrain, J., S.C. Wray: TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture, Portland. Lecture Notes in Computer Science Vol. 274, Springer-Verlag, Berlin 1987, pp. 34 – 45.

FIELD 1988 Field, A.J., P.G. Harrison: Functional Programming. Addison-Wesley, Wokingham 1988.

GRISS 1981 Griss, M.L., A.C. Hearn: A Portable LISP Compiler. Software — Practice and Experience Vol. 11 (1981), pp. 541 – 605.

HENDERSON 1980 Henderson, P.: Functional Programming: Application and Implementation. Prentice-Hall, London 1980.

HENSON 1987 Henson, M.C.: Elements of Functional Languages. Blackwell Scientific Publications, Oxford 1987.

HUGHES 1989 Hughes, J.: Why Functional Programming Matters. Computer Journal Vol. 32 (1989) No. 2, pp. 98 – 107.

JOHNSSON 1987 Johnsson, T.: Compiling Lazy Functional Languages. Ph.D. Thesis. Chalmers University of Technology, Department of Computer Sciences, Göteborg 1987.

LANDIN 1964 Landin, P.J.: The Mechanical Evaluation of Expressions. Computer Journal Vol. 6 (1964) No. 4, pp. 308 – 320.

MACQUEEN 1984 MacQueen, D.B., G.P. Plotkin, R. Sethi: An Ideal Model of Recursive Types. Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages. Salt Lake City, Utah 1984, pp. 165 – 174.

MILNER 1978 Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Science Vol. 17 (1978), pp. 348 – 375.

MYCROFT 1980 Mycroft, A.: The Theory and Practice of Transforming Call-by-need into Call-by-value. Proceedings of the Fourth „Colloque International sur la Programmation“, Paris. Lecture Notes in Computer Science Vol. 83, Springer-Verlag, Berlin 1980, pp. 269 – 281.

PEYTON JONES 1987 Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs, New Jersey 1987.

PEYTON JONES 1989 Peyton Jones, S.L.: Parallel Implementation of Functional Programming Languages. Computer Journal Vol. 32 (1989) No. 2, pp. 175 – 186.

Revesz 1988 Revesz, G.E.: Lambda-Calculus, Combinators, and Functional Programming. Cambridge University Press, Cambridge 1988.

Reynolds 1985 Reynolds, J.C.: Three Approaches to Type Structure. Mathematical Foundations of Software Development (Vol. 1). Lecture Notes in Computer Science Vol. 185, Springer-Verlag, Berlin 1985, pp. 97 – 138.

Turner 1979 Turner, D.A.: A New Implementation Technique for Applicative Languages. Software — Practice and Experience Vol. 9 (1979) No. 1, pp. 31 – 49.

Turner 1985 Turner, D.A.: Miranda: A Non-strict Functional Language with Polymorphic Types. Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy. Lecture Notes in Computer Science Vol. 201, Springer-Verlag, Berlin 1985, pp. 1 – 16.

Turner 1987 Turner, D.A.: An Introduction to Miranda. Appendix to Peyton Jones: The Implementation of Functional Programming Languages [Peyton Jones 1987].

Wikström 1987 Wikström, A.: Functional Programming Using Standard ML. Prentice-Hall, London 1987.

Wray 1989 Wray, S.C., J. Fairbrain: Non-strict Languages - Programming and Implementation. Computer Journal Vol. 32 (1989) No. 2, pp. 142 – 151.