Implementierung eines Übersetzers zur Erzeugung von effizientem C–Code aus funktionalem STG–Code

Frank Kiesel

Diplomarbeit
5. April 1995

eingereicht bei

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz / Softwaretechnologie

Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20)

Hiermit bestätige ich, daß ich die vorliegende Arbeit selbständig verfaßt habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 5. April 1994

Frank Kiesel

Danksagung

Ich möchte mich hiermit bei allen Personen bedanken, die mich bei der Anfertigung dieser Arbeit unterstützt haben. Mein besonderer Dank gilt Sven-Erik Panitz und Prof. Dr. Manfred Schmidt-Schauß für ihre ausgezeichnete Betreuung und ihre wertvollen Anregungen.

Frank Kiesel

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Funktionale Programmiersprachen gewinnen zunehmend an Bedeutung. Dafür gibt es viele Gründe. So gestatten sie z. B. eine viel klarere und kürzere Darstellung von Algorithmen als prozedurale Sprachen, besitzen keine Seiteneffekte und erlauben die Darstellung und Manipulation unendlicher Datenstrukturen. Außerdem sind funktionale Sprachen nicht *inhärent sequentiell*, also auch für parallele Computer geeignet. Die Parallelisierung funktionaler Programme kann sogar automatisch erfolgen.

Trotzdem werden funktionale Sprachen heute fast nur in der Forschung, wie z. B. in der künstlichen Intelligenz verwendet, wo Anwendungen oft aus komplexen Algorithmen zur Manipulation von großen, meist symbolischen Datenstrukturen bestehen. Eine allgemeine Verbreitung von funktionalen Sprachen fand bisher nicht statt, da bisher die Implementierung dieser Sprachen, im Vergleich zu gewöhnlichen Hochsprachen, nur sehr unbefriedigend gelang. Die Forschung in den letzten Jahren konzentrierte sich deshalb auf eine möglichst effiziente Implementierung von funktionalen Sprachen und brachte einige, durchaus mit anderen Hochsprachen konkurrenzfähige Implementierungen hervor.

In der vorliegenden Arbeit wird die Implementierung eines Übersetzers für eine funktionale Sprache beschrieben, wobei alle Aspekte der Implementierung betrachtet und erörtert werden. Den Ausgangspunkt für diese Implementierung bildet die von Peyton Jones entwickelte abstrakte *Spineless Tagless G–Machine* [Pey92].

1.1 Überblick

Kapitel 2 faßt kurz grundlegende Eigenschaften funktionaler Programmiersprachen zusammen und beschreibt die einzelnen Phasen der Übersetzung funktionaler Programme. Dabei findet eine genaue Einordnung dieser Arbeit statt (s. Abbildung 2.2 Seite 9), die ausgehend von einer einfachen, rein funktionalen Sprache (der STG–Sprache), die als funktionaler Zwischencode dient, C–Code erzeugt.

Abschnitt 2.3 beschreibt dann Eigenschaften der zu übersetzenden STG–Programme, die durch vorangegangene Phasen der Übersetzung als gegeben betrachtet werden können (die Programme sind korrekt getypt, das Überladen von Operatoren ist aufgelöst, Pattern Matching wurde in `case`–Ausdrücke transformiert, usw.).

In den Kapiteln 3 und 4 werden die STG–Sprache und die STG–Maschine beschrieben.

Die STG–Syntax wird dabei, außer um einige einfache Sprachkonstrukte wie z. B. Felder, auch um eine explizite Markierung von *unboxed values* (d. h. Daten eines einfachen Typs (Integer, Float, ...), die im Gegensatz zu anderen Implementierungen funktionaler Programmiersprachen in der STG–Maschine nicht in Datenstrukturen gehalten, sondern direkt manipuliert werden können) erweitert. Diese Markierungen ermöglichen die Definition polymorpher Funktionen zur Manipulation von unboxed values und erlauben die Verwendung weitaus effizienterer Datenstrukturen (*primitive Listen* — s. Abschnitt 3.2.6).

Die STG–Maschine wird um zwei Komponenten (den B–Stack und die Return–Register) erweitert, um die explizite Manipulation von unboxed values schon auf dieser Ebene deutlich zu machen und somit die spätere Codeerzeugung zu vereinfachen.

Danach wird in den Kapiteln 5 und 6 die Umsetzung der Komponenten der abstrakten Maschine und der STG–Programme in C–Code beschrieben. Dabei werden Implementierungen anderer funktionaler Sprachen betrachtet und verglichen.

In Kapitel 7 wird schließlich noch die Realisierung einiger Komponenten der Laufzeitumgebung beschrieben, die unabhängig von der abstrakten Maschine sind (Garbage Collection, Tail Call Optimierung, ...).

Kapitel 8 faßt die vorliegende Arbeit zusammen, diskutiert die Resultate anhand des Vergleichs einiger Beispielprogramme mit anderen Übersetzern und gibt einen Ausblick.

Kapitel 2

Grundlagen

2.1 Funktionale Sprachen

Höhere Programmiersprachen abstrahieren in verschiedenem Maße von der Struktur und den Details der Rechner, auf denen die geschriebenen Programme ausgeführt werden sollen. Imperative Programmiersprachen z. B. orientieren sich sehr stark an der Struktur der „von Neumann"–Architektur der Rechner. So spiegeln sich Strukturen und Operationen wie Zeiger, Sprünge, linearer Speicher, indizierter Zugriff usw. teilweise in den Konzepten dieser Sprachen wider. Imperative Programme beschreiben eine Folge von Instruktionen, die Schritt für Schritt abgearbeitet werden und dabei den Zustand von Variablen (benannte Speicherzellen) verändern oder den Kontrollfluß steuern. Sie beschreiben also eine sequentielle Lösung eines Problems, auch wenn dessen Teilprobleme in keinem Abhängigkeitsverhältnis zueinander stehen.

Funktionale Sprachen (wie z. B. Haskell [Hud+92] oder Clean [SNGP91]) hingegen erlauben eine Programmierung auf einer wesentlich höheren Ebene als imperative Sprachen, da sie sehr viel stärker von der Struktur der realen Rechner abstrahieren. Am deutlichsten wird dies bei Variablen, die im Gegensatz zu imperativen Sprachen keine Speicherzellen darstellen, deren Wert während der Programmausführung beliebig verändert werden kann. Vielmehr stellt eine Variable in einem funktionalen Programm einen Bezeichner für einen Ausdruck dar, dessen Wert sich nicht verändern kann. Lediglich die Reduktion des von ihr bezeichneten Ausdrucks auf dessen Wert ist möglich.

Das Ausführungsprinzip funktionaler Sprachen ist die **Reduktion**, d. h. ein funktionales Programm, das neben Funktionsdefinitionen auch einen Ausdruck enthält, der den Wert des Programms beschreibt, wird ausgewertet, indem in einzelnen Schritten so lange jeweils ein (Teil–) Ausdruck durch eine äquivalente einfachere Form ersetzt wird, bis der Ausdruck keine reduzierbaren Teilausdrücke mehr enthält (**Normalform**). Die einzelnen **Reduktionsschritte** können durch Term– bzw. Graphersetzung beschrieben werden. Die Reduktion des Ausdrucks `f 5 (3+4)`, wobei `f x y = x * y` gilt, kann wie folgt dargestellt werden:

$$\texttt{f 5 (3+4)} \rightarrow \texttt{f 5 7} \rightarrow \texttt{5 * 7} \rightarrow \texttt{35}$$

Dabei wird die Auswertungsreihenfolge, mit der die Teilausdrücke reduziert werden, nicht vom Programmierer vorgegeben. Da die Funktionen keinerlei Seiteneffekte besitzen (etwa die Manipulation globaler Variablen), ist der Wert jeder Funktion (also ihre Normalform) ausschließlich von ihren Argumenten abhängig, gleichgültig zu welchem Zeitpunkt sie reduziert wird (**referentielle Transparenz**). Somit spielt die Reihenfolge, in der die Teilausdrücke ausgewertet werden, im Prinzip keine Rolle. Allerdings existieren auch Ausdrücke, die keine Normalform besitzen (z.B. 1/0). Dies bedeutet, daß die Reduktion eines solchen Ausdrucks nicht fehlerfrei terminiert. Enthält ein Ausdruck, der eine Normalform besitzt, einen Teilausdruck ohne Normalform, so führt eine Reduktionsfolge, die zunächst diesen Teilausdruck betrachtet, nicht zum gewünschten Ergebnis. Z. B. besitzt der Ausdruck

```
K 5 (1/0)
```

mit

```
K x y = x
```

die Normalform 5. Eine Reduktionsfolge, die zunächst den Teilausdruck `(1/0)` reduziert, wird diese Normalform jedoch nicht finden. Allerdings kann man beweisen, daß die sog. **Normal Order Reduction**, bei der immer der äußerste, am weitesten links stehende, reduzierbare Teilausdruck zuerst reduziert wird, eine Reduktionsfolge beschreibt, die immer die Normalform eines Ausdrucks (falls vorhanden) findet (**Church-Rosser–Theorem**)[1]. Im obigen Beispiel bedeutet dies, daß zunächst die Funktion `K` angewandt wird, wodurch sich bereits die Normalform 5 ergibt. Die Normal Order Reduction bewirkt also, im Gegensatz zu strikten Sprachen, bei denen die Argumente immer in ausgewerteter Form übergeben werden (call-by-value), eine Übergabe von Argumenten an Funktionen in unausgewerteter Form und wird auch als **call-by-name**–Parameterübergabemechanismus bezeichnet. Die Normal Order Reduction hat den Vorteil, daß Argumente von Funktionen nur dann ausgewertet werden, wenn sie tatsächlich von der Funktion benötigt werden. Allerdings kann es vorkommen, daß Argumente bei wiederholter Verwendung mehrfach ausgewertet werden. Dies wird vermieden, indem gleiche Teilausdrücke identifiziert werden[2], wodurch ein Teilausdruck nur bei seiner ersten Verwendung reduziert werden muß. Bei jeder weiteren Verwendung

[1] Das Church-Rosser–Theorem I beweist die Eindeutigkeit der Normalform, Theorem II beweist, daß die Normal Order Reduction diese Normalform (falls vorhanden) immer findet. Beweise hierzu finden sich in [Wel75] und [Ros82].

[2] Durch diese Identifikation von Teilausdrücken werden bei der graphischen Darstellung dieser Ausdrücke aus Bäumen Graphen. Die Auswertung dieser Ausdrücke wird deshalb auch Graphreduktion genannt.

des Teilausdrucks kann dann dessen reduzierte Form verwendet werden. Diese Optimierung wird im allg. als **call-by-need** oder **lazy evaluation** bezeichnet.

Diese Auswertungsstrategie erlaubt dem Programmierer auch kostspielige Berechnungen als Argumente an Funktionen zu übergeben, ohne sich darüber Gedanken machen zu müssen, ob das Ergebnis tatsächlich benötigt wird. Desweiteren gestattet sie auch eine einfache Handhabung von Datenobjekten, die ausgewertet sehr groß, evtl. sogar unendlich groß sind, weil auch Datenobjekte nur so weit ausgewertet werden, wie dies für die Programmauswertung erforderlich ist. Diese beiden Aspekte tragen zu wesentlich klareren und übersichtlicher strukturierten Programmen bei, als dies bei einer Auswertungsstrategie wie z. B. call-by-value möglich ist.

2.2 Implementierung funktionaler Sprachen

Compiler für funktionale Sprachen bestehen, genau wie Compiler anderer Sprachen aus zwei Teilen. Der erste Teil, auch **Front End** genannt, führt hardware–unabhängige *source-to-source* Transformationen durch, während der zweite Teil, das **Back End**, die Anpassung an die Hardware vornimmt, also den ausführbaren Code erzeugt (Abb. 2.2).

2.2.1 Das Front End

Das Front End besteht aus einem Scanner und einem Parser und einer Reihe von Transformationen, die dem funktionalen Programm Informationen hinzufügen, es vereinfachen und optimieren. Zu diesen Transformationen zählen:

- eine Vereinfachung (Desugaring)
- eine Abhängigkeitsanalyse (Dependancy analysis)
- eine Typüberprüfung (Type checking)
- und Optimierungen wie z. B. Striktheitsanalyse (Strictness analysis)

2.2.1.1 Scanner und der Parser

Der Scanner und der Parser wandeln ein als Textfile gegebenes Programm in eine für die weitere Verarbeitung geeignete Darstellung, den sog. **Syntaxbaum** um. Hierzu wird zunächst der Text vom Scanner in einzelne Token zerlegt (**lexical analysis**), die dann zum Syntaxbaum zusammengesetzt werden. Während dieses Vorgangs können evtl. im Programm vorhandene Syntaxfehler erkannt werden.

2.2.1.2 Vereinfachung

Durch diese Transformation werden leistungsfähige Sprachkonstrukte, die dem Programmierer eine „bequeme“ Darstellung seiner Algorithmen erlauben, in wesentlich einfachere (für die nachfolgenden Transformationen und den Code–Generator leichter zu verarbeitende) Konstrukte umgewandelt. Zu diesen Sprachkonstrukten gehören z. B.

- Listenausdrücke, die eine bequeme Verarbeitung von Listen erlauben. So gestatten z. B. Ausdrücke wie

  ```
  [ x | x <- list, p x ]
  ```

  die Bezeichnung einer Liste, die alle Elemente aus `list` enthält, die das Prädikat `p` erfüllen. Solche Ausdrücke werden in einfache (rekursive) Funktionen umgewandelt.

- Guards und Pattern Matching, die es erlauben Funktionen über Fälle zu definieren. D. h. es werden für verschiedene Werte, die ein Argument einer Funktion annehmen kann, verschiedene Funktionsdefinitionen angegeben. Funktionen, die diese Möglichkeiten verwenden, werden in Funktionen umgewandelt, die zunächst das entsprechende Argument betrachten und dann in Abhängigkeit seines Wertes verzweigen.

2.2.1.3 Abhängigkeitsanalyse

Während dieser Analyse werden Abhängigkeiten der im Programm definierten Funktionen untereinander bestimmt. Die Abhängigkeitsanalyse besteht im wesentlichen aus einer Umordnung der im Programm enthaltenen Funktionsdefinitionen, so daß die Definitionen vor den Funktionsanwendungen im Programm erscheinen. Diese Umordnung dient der Fehlererkennung (z. B. der Verwendung einer nicht definierten Funktion) und der Vereinfachung der nachfolgenden Transformationen (insbesondere der Typüberprüfung).

2.2.1.4 Typüberprüfung

Anders als bei prozeduralen Sprachen müssen bei funktionalen Sprachen die Typen von Funktionen oder Variablen nicht angegeben werden. Für alle Objekte, die nicht vollständig durch den Programmierer getypt wurden, wird durch einen **Typinferenzalgorithmus** anhand der Funktionsdefinition und der Funktionsanwendung das allgemeinste Typschema hergeleitet. Damit kann überprüft werden, ob alle Ausdrücke typkorrekt aufgebaut sind. Außerdem können die ermittelten Typinformationen dem Code–Generator für Optimierungen zur Verfügung gestellt werden (s. Abschnitt 5.6.4).

2.2.1.5 Striktheitsanalyse

Striktheit ist im Prinzip das Gegenteil von laziness. Bei **lazy funktionalen Sprachen** (dementsprechend auch **nicht-strikte funktionale Sprachen** genannt) werden Funktionsargumente in unausgewerteter Form übergeben (s. o.). Diese Art der Argumentübergabe (call-by-name bzw. call-by-need) ist jedoch mit einem großen Aufwand für die Erzeugung und Manipulation der unausgewerteten Funktionsargumente verbunden. Wenn bekannt ist, daß eine Funktion eines ihrer Argumente sicher auswertet, so kann dieses Argument ohne weiteres vor der Übergabe ausgewertet und in dieser Form an die Funktion übergeben werden. Damit wird die Darstellung dieser Argumente in unausgewerteter Form gespart. Eine generelle Übergabe der Argumente in ausgewerteter Form (wie bei strikten Sprachen) kann Ausdrücke wie z. B. `K 5 (1/0)` (im Bsp. oben) nicht reduzieren und wird deshalb nicht weiter betrachtet.

Die Bestimmung von Argumenten, die bereits in ausgewerteter Form an Funktionen übergeben werden können, bezeichnet man als **Striktheitsanalyse**. Striktheitsanalyse erfolgt mit Hilfe der abstrakten Interpretation oder der abstrakten Reduktion, die im Vergleich zur abstrakten Interpretation wesentlich effizienter umzusetzen ist (vgl. hierzu Nöcker [Nöc93] und Schütz [Sch94]). Die Optimierung eines Programms mit Hilfe von Striktheitsinformationen kann sowohl die Laufzeit, als auch den Speicherbedarf eines Programms wesentlich verringern. Wray [FW87] gibt Beispiele an, bei denen die Laufzeit auf ein Drittel reduziert werden kann, Burton [BMOR87] gibt Beispiele, bei denen der Platzbedarf um eine Größenordnung reduziert werden kann.

2.2.1.6 Zwischencodegenerator

Da funktionale Sprachen nicht so klar wie imperative Sprachen die von-Neumann–Architektur widerspiegeln, führt man sog. **abstrakte Maschinen** zur Graphreduktion ein, die auf diesen Sprachtyp besser angepaßt sind. Abstrakte Maschinen beschreiben die Semantik von Programmiersprachen, z. B. durch Zustandsübergänge der Maschine bei verschiedenen Eingaben (operationale Semantik). Sie unterstützen die Konzepte funktionaler Sprachen besser als reale Maschinen und vereinfachen deshalb die Implementierung. Eine der ersten abstrakten Maschinen zur Erklärung der Übersetzung funktionaler Programmiersprachen war die SECD-Maschine von Landin [Lan64]. Sie wurde entwickelt, um die Semantik für Lisp–Programme mit statischer Bindung und Call-by-Value–Parameterübergabe zu definieren. Die von Johnsson [Joh84] und Augustsson [Aug84] entwickelte G-Maschine ist die erste abstrakte Maschine mit programmierter Graphreduktion. D. h. die Ausdrücke in einem funktionalen Programm werden in sog. G-Code übersetzt, der dann zur Laufzeit Instanzen der Ausdrücke erzeugt, was eine besonders schnelle Graphreduktion ermöglicht. Peyton Jones' [Pey92] Spineless-Tagless-G-Maschine (kurz STG-Maschine) stellt eine Weiterentwicklung der G-Maschine dar und bildet die Grundlage dieser Arbeit.

Der Zwischencodegenerator erzeugt nun den „Maschinencode“ für die abstrakte Maschine, der im allg. nicht mehr funktional ist. Eine Ausnahme bildet hier die STG-Maschine, deren Maschinencode eine einfache rein-funktionale Sprache, die sog. STG–Sprache ist.

2.2.2 Das Back End

Das Back End nimmt die Anpassung an die reale Maschine vor, indem der abstrakte Maschinencode in realen Maschinencode umgewandelt wird. Die abstrakte Maschine wird nur einmal an die Hardware angepaßt und stellt die Laufzeitumgebung (runtime module) dar, die die Ausführung des erzeugten Maschinencodes steuert.

Dabei hat es sich als vorteilhaft erwiesen, nicht direkt realen Maschinencode zu erzeugen, sondern C–Code. Die Programmiersprache C steht auf fast allen Maschinen zur Verfügung und dient als universelle Zwischensprache. Der C-Compiler stellt also letzlich das eigentliche Back End dar und erzeugt den ausführbaren Code.

2.3 Eigenschaften der zu übersetzenden STG–Programme

Nach den Transformationen, die das zu übersetzende Funktionale Programm im Front-End durchlaufen hat (Abb. 2.2 S. 9), besitzt der resultierende funktionale Zwischencode (d. h. das für den hier betrachteten Übersetzer als Eingabe dienende STG–Programm) einige Eigenschaften, die hier kurz beschrieben werden:

- Das zu übersetzende Programm ist korrekt getypt.
- Das Überladen von Operatoren (eine Fähigkeit der meisten funktionalen Sprachen) ist bereits aufgelöst.
- Das Programm enthält keine „leistungsfähigen“ Sprachkonstrukte wie z. B. Listenausdrücke oder Guards mehr. Diese sind bereits in einfachen funktionalen Code transformiert.
- Pattern Matching ist bereits in `case`–Ausdrücke umgewandelt.
- Speziell auf die STG–Sprache abgestimmte Optimierungen sind bereits ausgeführt (z. B. Bestimmung des Update–Flags, s. Abschnitt 3.1).

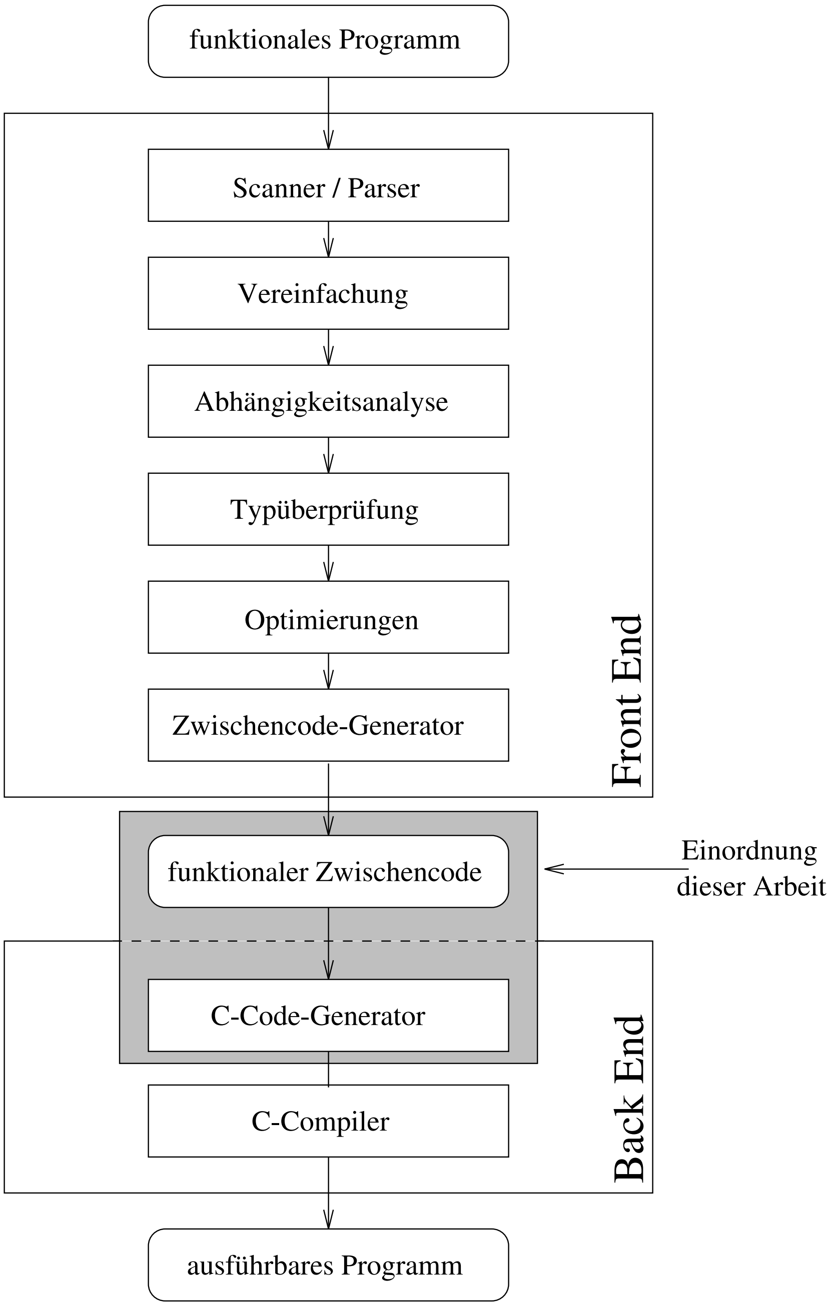


Abbildung 2.1: *Ablauf der Übersetzung funktionaler Programme*

Kapitel 3

Syntax

Der abstrakte Maschinencode für die Spineless Tagless G–Maschine ist eine einfache, rein–funktionale Sprache, die im weiteren als **STG–Sprache** bezeichnet wird. Die hier verwendete STG–Sprache besitzt im Vergleich zu der in Peyton Jones [Pey92] definierten Sprache einige Ergänzungen und Erweiterungen, die eine vereinfachte Übersetzung in C–Code ermöglichen und einige zusätzliche Sprachkonstrukte, wie z. B. Felder und Stringkonstanten gestatten.

3.1 Die funktionale STG–Sprache

Abb. 3.1 und 3.2 zeigen die Syntax der hier verwendeten Sprache in Backus–Naur–Form (BNF), die folgende Eigenschaften hat:

- Alle Funktions- und Konstruktorargumente sind einfache Variablen oder Konstanten. D. h. komplexere Ausdrücke werden vor der Funktions- oder Konstruktoranwendung mit einem `let(rec)`–Ausdruck im Heap erzeugt oder ausgewertet.
- Alle Konstruktoren (s. Abschnitt 3.2.1 und interne Funktionen (Addition, Multiplikation, ...) sind gesättigt. D. h. diese Funktionen und alle Konstruktoren werden immer nur auf eine ausreichende Anzahl von Argumenten angewendet.
- Pattern Matching wird nur mit Hilfe der `case`–Anweisung durchgeführt, wobei die verwendeten Patterns einfache 'one level' Patterns sind (zur Umsetzung komplexerer Patterns in diese Form s. [Pey87] S. 78–103).
- Die STG–Sprache hat eine spezielle Form der Bindung:
  $$f = \{v_1,\ \ldots, v_n\}\ \backslash\pi\ \{x_1,\ \ldots,\ x_m\}\ \texttt{->}\ e$$
  Die rechte Seite dieser Bindung wird als **lambda-form** bezeichnet und besteht aus
  - freien Variablen v_i,

- einem Update–Flag $\backslash\pi$,
- Argumenten x_i und
- einem Ausdruck e, der an den Namen f gebunden wird.

Eine solche Bindung kann auf zwei Arten interpretiert werden. Aus einer denotationellen Sicht wird der Ausdruck $(\lambda x_1 \dots x_m.e)$ an den Bezeichner f gebunden. Aus der operationalen Sicht ist f ein im Speicher zu haltendes Datenobjekt, das einen Zeiger auf ausführbaren Code und (Zeiger auf) die freien Variablen v_i enthält und die Funktion $(\lambda x_1 \dots x_m.e)$ repräsentiert.

Die Verwendung von freien Variablen ist eine Besonderheit der STG–Sprache und erspart bei der Transformation einer funktionalen Sprache, wie z. B. Haskell [Hud$^+$92] in die STG–Sprache ein **lambda-lifting** — eine z. B. bei der Verwendung der G-Maschine notwendige Programmtransformation, die Ausdrücke mit freien Variablen in eine zur Erzeugung von G-Code geeignete Form (sog. supercombinators) umwandelt (vgl. hierzu [Pey87] Kapitel 13).

Das **Update–Flag** π, das den Wert `u` (für updatable) oder `n` (für non–updatable) annehmen kann, gibt an, ob der Ausdruck f nach seiner Reduktion durch sein Ergebnis ersetzt werden soll. Mit Hilfe dieses Update–Flags kann sowohl der call-by-name– als auch der call-by-need–Parameterübergabemechanismus realisiert werden. Werden z. B. alle Update–Flags auf `\n` gesetzt, so wird kein Ausdruck nach seiner Reduktion durch seinen Wert ersetzt (call-by-name). Eine wiederholte Verwendung eines Ausdrucks führt also zu einer erneuten Reduktion. Werden hingegen alle Update–Flags auf `\u` gesetzt, so werden alle Ausdrücke nach ihrer Reduktion durch ihren Wert ersetzt, der bei einer weiteren Verwendung des Ausdrucks benutzt werden kann (call-by-need). Die explizite Angabe dieses Flags für jeden Ausdruck getrennt, gestattet es im Gegensatz zur G–Maschine, bei der nach fast jeder Reduktion ein Update stattfindet, nur solche Ausdrücke durch ihren Wert zu ersetzen, die tatsächlich mehr als einmal verwendet werden.

Die genaue Bestimmung des Update–Flags stellt eine Optimierung dar, die vor der Erzeugung des Zwischencodes, also der STG–Sprache, erfolgt und wird hier nicht weiter betrachtet (siehe hierzu [Rest] und [Pey92]).

- Die STG–Sprache unterstützt die Verwendung von **unboxed values** — im weiteren **primitive Werte** genannt (s. Abschnitt 3.2.4).

<table>
<tr><td>Program</td><td>prog</td><td>→</td><td>binds</td></tr>
<tr><td>Bindings</td><td>binds</td><td>→</td><td>var “=” lf { var “=” lf }</td></tr>
<tr><td>Lambda Forms</td><td>lf</td><td>→</td><td>vars “\”upd vars “->” expr</td></tr>
<tr><td>Update Flag</td><td>upd</td><td>→</td><td>“u” | “n”</td></tr>
<tr><td>Expression</td><td>expr</td><td>→</td><td>“let” binds “in” expr</td></tr>
<tr><td></td><td></td><td>|</td><td>“letrec” binds “in” expr</td></tr>
<tr><td></td><td></td><td>|</td><td>“case” expr “of” alts “esac”</td></tr>
<tr><td></td><td></td><td>|</td><td>var [atoms]</td></tr>
<tr><td></td><td></td><td>|</td><td>constr [atoms] | constr “[” [“*”] int “]”</td></tr>
<tr><td></td><td></td><td>|</td><td>prim atoms</td></tr>
<tr><td></td><td></td><td>|</td><td>literal</td></tr>
<tr><td>Alternatives</td><td>alts</td><td>→</td><td>“{” int “}”</td></tr>
<tr><td></td><td></td><td></td><td>(aalts | adefault) { (aalts | adefault) }</td></tr>
<tr><td></td><td></td><td>|</td><td>(palts | pdefault) { (palts | pdefault) }</td></tr>
<tr><td>Algebraic Alt</td><td>aalt</td><td>→</td><td>constr [vars] “->” expr</td></tr>
<tr><td>Primitive Alt</td><td>palt</td><td>→</td><td>literal “->” expr</td></tr>
<tr><td>Default Aalt</td><td>adefault</td><td>→</td><td>var “->” expr</td></tr>
<tr><td></td><td></td><td>|</td><td>“default ->” expr</td></tr>
<tr><td>Default Palt</td><td>pdefault</td><td>→</td><td>var “->” expr</td></tr>
<tr><td>Constructors</td><td><i>constr</i></td><td>→</td><td>“<”<i>int</i> “,” <i>int</i>“>”</td></tr>
<tr><td>Primitive Ops</td><td>prim</td><td>→</td><td>“+” | “-” | “*” | “/” |
“%” | “:>” | “:<” | “==” | ...</td></tr>
<tr><td>Variable List</td><td>vars</td><td>→</td><td>“{” [var { “,” var}] “}”</td></tr>
<tr><td>Atom List</td><td>atoms</td><td>→</td><td>“{” [atom { “,” atom}] “}”</td></tr>
<tr><td>Atoms</td><td>atom</td><td>→</td><td>var | literal</td></tr>
<tr><td>Literals</td><td>literal</td><td>→</td><td>[“+” | “-”] int</td></tr>
<tr><td></td><td></td><td>|</td><td>[“+” | “-”] float</td></tr>
<tr><td></td><td></td><td>|</td><td>“'”charconst“'”</td></tr>
<tr><td></td><td></td><td>|</td><td>“"”stringconst“"”</td></tr>
<tr><td></td><td></td><td>|</td><td>hexconst</td></tr>
</table>

Abbildung 3.1: Die Syntax der STG–Sprache in Backus–Naur–Form

Integer constant	*int*	→	*digit* { *digit* }
Digit	*digit*	→	“0” $\mid$ “1” $\mid$ … $\mid$ “9”
Floats	*float*	→	(*int*“.” $\mid$ *int*“.”*int* $\mid$ “.”*int*)
			[(“e”$\mid$“E”)[“+” $\mid$ “-”] *int*]
Variable	*var*	→	*varstart* {(*digit* $\mid$ *letter* $\mid$ “_”)}[“#”]
Variable Start	*varstart*	→	*letter* $\mid$ “_”
Letter	*letter*	→	“a” $\mid$ … $\mid$ “z” $\mid$ “A” $\mid$ … $\mid$ “Z”
Character	*charconst*	→	*letter*
		$\mid$	“\b” $\mid$ “\t” $\mid$ “\n” $\mid$ “\f”
		$\mid$	“\r” $\mid$ “\\” $\mid$ “\'” $\mid$ “\0”
Hex const	*hexconst*	→	(“0x” $\mid$ “0X”) *hexdigit hexdigit*
Hex digit	*hexdigit*	→	(*digit* $\mid$ *hexletter*)
Hex letter	*hexletter*	→	“a” $\mid$ “b” $\mid$ “c” $\mid$ “d” $\mid$ “e” $\mid$ “f”
String	*stringconst*	→	{ *letter* $\mid$ *digit* $\mid$ *charconst* $\mid$ *special* }
Special character	*special*	→	“.” $\mid$ “,” $\mid$ “-” $\mid$ …

Abbildung 3.2: Die Syntax der STG–Sprache in Backus–Naur–Form (Fortsetzung)

3.2 Erweiterungen und Ergänzungen der Syntax

3.2.1 Konstruktoren

Funktionale Sprachen wie Haskell [Hud+92] gestatten dem Programmierer außer der Verwendung von Standarddatentypen wie `Int`, `Float`, `String` usw. auch die Definition problembezogener Datentypen, indem für jeden Typ eine Menge von **Konstruktoren** bestimmter Stelligkeit und deren Operandentypen angegeben werden. In Haskell–Notation beschreibt z. B.

```
data Tree = Branch Tree Tree
          | Leaf Int
```

eine Datentyp `Tree` zur Darstellung von Bäumen. Objekte dieses Typs werden dann durch die Anwendung der Konstruktoren `Branch` bzw. `Leaf` auf entsprechende Operanden erzeugt (z. B. `Leaf 5`). Diese Datentypen werden im weiteren als **algebraische Datentypen** bezeichnet.

Bei der Übersetzung funktionaler Sprachen, also auch in der Zwischensprache, werden Konstruktoren nicht mehr durch ihre Namen (`Branch` bzw. `Leaf`) dargestellt. Vielmehr werden alle Konstruktoren eines Typs mit ganzen Zahlen (bei der STG–Sprache beginnend bei Null) durchnumeriert. Diese den Konstruktor darstellende ganze Zahl wird als **Konstruktortag** bezeichnet.

Im Gegensatz zu der in [Pey92] beschriebenen STG–Sprache, werden die Konstruktoren hier durch zwei Integer–Konstanten in spitzen Klammern dargestellt (s. Syntax Abb. 3.1: *constr*). Die erste der beiden Integer–Konstanten stellt das Konstruktortag dar. Die zweite, zusätzliche Integer–Konstante dient der Ausgabefunktion als Markierung, um den Konstruktor und dessen Komponenten in der richtigen Form zu behandeln (d. h. auszugeben oder auszuwerten) und wird im weiteren als **Ausgabemarkierung** bezeichnet.

Die Konstruktortags dienen zum Aufbau einer Sprungtabelle für die Alternativen eines `case`–Ausdrucks. Diese Tabelle enthält für jede mögliche Alternative einen Zeiger auf den Code der diese Alternative ausführt. Dies ermöglicht den Zugriff auf die Alternativen in konstanter Zeit. Zum Aufbau dieser Tabelle wird jedoch die Anzahl der möglichen Alternativen benötigt (diese entspricht der Anzahl der Konstruktoren des im `case`–Ausdrucks zu betrachtenden Typs). Da diese Anzahl nicht notwendigerweise aus dem STG–Programm hervor geht (insbesondere bei der Zusammenfassung mehrerer Alternativen zu einer `default`–Alternative), wird diese Anzahl zu Beginn der Alternativen in geschweiften Klammern angegeben (s. Abb. 3.1: *alts*).

Sowohl diese Anzahl als auch die Ausgabemarkierung stellen Typinformationen dar, die zur einfacheren Weiterverarbeitung in die Syntax eingefügt werden.

3.2.2 Felder

Felder erlauben im Gegensatz zu Listen (ein in allen funktionalen Sprachen vorhandener Datentyp zur Darstellung einfach verketteter Listen) den Zugriff auf ihre Elemente in konstanter Zeit. Allerdings besitzen sie eine konstante, zur Laufzeit nicht veränderbare Größe.

Felder werden genau wie Konstruktoren durch zwei Integer–Konstanten dargestellt. Die Definition eines Feldes kann auf zwei Arten erfolgen:

- durch implizite Initialisierung, indem die Werte der Feldelemente in der Liste der freien Variablen angegeben werden[1]:

  ```
  array1 = {1, 2, 3} \n {} -> <0,7> {array1}
  ```

  `array1` stellt ein Feld der Länge 3 dar, wobei die Feldelemente die Werte 1, 2, bzw. 3 besitzen. Die Feldelemente sind dabei nicht auf Konstanten

[1] dies führt zu einer ebenfalls impliziten Initialisierung im entstehenden C–Programm (vgl. Kernighan, Ritchie [KR83]), die keine Auswirkung auf die Laufzeit hat

beschränkt, sondern können den Wert jedes im Geltungsbereich vorhandenen Namens annehmen[2].

- durch die Angabe der Feldgröße hinter dem Konstruktor. Dabei kann es sich um eine Integer–Konstante oder eine Variable handeln, die eine ganze Zahl beschreibt. Die Feldgröße wird in eckigen Klammern angegeben und optional durch einen * eingeleitet. Dieser * zeigt an, daß im Falle einer Garbage–Collection die Feldelemente als Zeiger auf Heap–Objekte interpretiert und verfolgt werden müssen. Dies ist notwendig, da im Gegensatz zur impliziten Initialisierung keine Feldelemente vorhanden sind, durch die der Typ festgelegt ist.

  ```
  array2 = {} \n {} -> <0,7> [100]

  array3 = {} \n {} -> <0,8> [*100]
  ```

  `array2` beschreibt ein Feld mit 100 Elementen primitiven Typs (d. h. Elemente, die bei einer Garbage–Collection nicht verfolgt werden müssen), während `array3` ein Feld gleicher Größe beschreibt, dessen Elemente Zeiger auf Heap–Objekte (z. B. noch nicht ausgewertete Ausdrücke) darstellen. Die Elemente beider Felder besitzen zu Beginn den Wert 0.

Der Zugriff auf die Elemente eines Feldes erfolgt mit den primitiven Funktionen `:<` zum Lesen und `:>` zum Schreiben. Die Lesefunktion `:<` besitzt zwei Argumente, wobei das erste der Name des Feldes ist und das zweite den Index darstellt. Das erste im Feld enthaltene Element besitzt den Index 1. So liefert z. B. folgende Funktion bei ihrer Auswertung den Wert des zweiten Elementes des oben definierten `array1`[3]:

```
f = {} \n {} -> :< {array1, 2}
```

Die Schreibfunktion hingegen ist dreistellig. Sie benötigt den Namen des Feldes, den Index des zu überschreibenden Elementes und den Wert, der eingefügt werden soll. Folgendes Beispiel ersetzt das dritte Element des oben definierten `array1` durch den Wert 9:

```
g = {} \n {} -> :> {array1, 3, 9}
```

[2] Um eine Liste durch die Ausgabefunktion (gesteuert durch die Ausgabemarkierung — im Bsp.: 7) ausgeben zu können, müssen alle Listenelemente den gleichen, dem Konstruktor entsprechenden Typ aufweisen. Im Prinzip ist es möglich, Daten unterschiedlichen Typs in ein Feld zu schreiben (sofern dies ein Typechecker zuläßt). Allerdings dürfen keine primitiven Daten zusammen mit Zeigern in einem Feld vorkommen, da dies zu Problemen bei der Garbage–Collection führt.

[3] Falls `array1` nicht auf oberster Ebene im Programm definiert ist, muß der Feldname `array1` in der Liste der freien Variablen der Funktion f vorkommen.

3.2.3 Erweiterte Literale

Desweiteren wurden Literale (s. Abb. 3.1: *literal*) um Fließkommazahlen, Zeichenketten und Zeichenkonstanten erweitert.

Fließkommazahlen müssen zu ihrer Identifikation einen Dezimalpunkt enthalten. Zusätzlich kann ein ganzzahliger Exponent, beginnend mit dem Zeichen `e` oder `E` und optionalem Vorzeichen angegeben werden. Zeichenketten werden zwischen zwei `"` angegeben und als Konstanten behandelt. Zeichenketten, bei denen einzelne Zeichen manipuliert werden sollen, sollten deshalb als Listen oder Felder realisiert werden.

Zeichenkonstanten werden in zwei einfache Anführungsstriche eingeschlossen und bestehen aus einem einzelnen Zeichen, also z. B. `'x'`. Zusätzlich können auch folgende Sonderzeichen mit Hilfe des Fluchtsymbols \ angegeben werden (s. Abb. 3.2: *charconst*):

backspace	BS	\b
Tabulatorzeichen	HT	\t
Zeilentrenner	LF	\n
formfeed	FF	\f
carriage return	CR	\r
Fluchtsymbol	\	\\
Anführungszeichen	'	\'
NULL–Zeichen	NUL	\0

Nicht druckbare Zeichen können als Hexadezimalkonstante, eingeschlossen in zwei einfache Anführungszeichen, dargestellt werden. Eine Hexadezimalkonstante (s. Abb. 3.2: *hexconst*) besteht aus `\x` gefolgt von zwei Hexadezimalziffern.

```
last = {} \n {} -> '\xff'
```

3.2.4 Kennzeichnung primitiver Werte (unboxed values)

Variablen in den Implementierungen nicht-strikter funktionaler Programmiersprachen bezeichnen in der Regel im Speicher gehaltene Datenstrukturen (**closures**), die (un-)ausgewertete Ausdrücke darstellen. Wird der Wert einer Variablen benötigt, so wird der entsprechende Ausdruck ausgewertet und die Datenstruktur mit dem Ergebnis überschrieben (call-by-need). Dies gilt auch für Variablen, die einen einfachen Datentyp, wie z. B. Integer besitzen. D. h. einfache Zahlen werden durch einen Zeiger auf eine Datenstruktur dargestellt, die die eigentliche Zahl enthält. Peyton Jones [Pey92] bezeichnet diese Datenstruktur als **box** oder **boxed value** und den darin enthaltenen Wert als **unboxed value** (im weiteren **primitiver Wert** genannt).

Diese Repräsentation primitiver Werte macht Arithmetik sehr teuer. Für eine einfache Addition z. B. müssen nach der Auswertung der beiden Operanden deren Werte aus den Datenstrukturen geholt und addiert werden. Anschließend muß eine neue Datenstruktur angelegt werden, die das Ergebnis aufnimmt. Die Ineffizienz dieser Darstellung wird besonders bei rekursiven arithmetischen Funktionen wie z. B. der Fakultätsfunktion deutlich, bei der für jedes Zwischenergebnis (Rückgabewert eines rekursiven Aufrufs) eine neue Datenstruktur angelegt werden muß.

Eine Besonderheit der STG–Sprache ist die Fähigkeit, primitive Werte (unboxed values) direkt zu manipulieren. Variablen können also nicht nur boxed values sondern auch unboxed values bezeichnen. Somit kann z. B. an arithmetische Funktionen der primitive Wert übergeben und weiterverarbeitet werden. Eine arithmetische Funktion wie z. B. die Fakultätsfunktion wird hierzu in zwei Funktionen, die sog. **Wrapper**– und die **Worker**–Funktion zerlegt. Die Wrapper–Funktion wertet das Argument aus und übergibt es als unboxed value an die Worker–Funktion. Die Worker–Funktion stellt die eigentliche Fakultätsfunktion dar, die nun mit unboxed values arbeiten kann.

```
fac_wrapper = {} \n {n} -> case n of
          Int {l#} -> case fac_worker {l#} of
                   m# -> Int {m#}
                   esac
          esac

fac_worker = {} \n {n#} -> case n# of
          0  -> 0
          m# -> case - {m#, 1} of
                k# -> case fac_worker {k#} of
                    l# -> * {m#, l#}
                    esac
                esac
          esac
```

In diesem Beispiel sind alle unboxed values mit einem `#` gekennzeichnet. Der Wrapper `fac_wrapper` wertet das Argument `n`, einen boxed value aus und übergibt den darin enthaltenen unboxed value `l#` an die Worker–Funktion `fac_worker` weiter. `fac_worker` kann nun die gesamte Rekursion auf unboxed values ausführen und liefert auch einen unboxed value als Ergebnis an den Wrapper zurück. Dieser unboxed value (`m#`) wird nun wieder in einen boxed value umgewandelt, indem der Konstruktor `Int` darauf angewandt wird (`Int {m#}`). (Eine genaue Beschreibung von unboxed values findet sich in [Pey91].)

Anders als in [Pey91] und [Pey92], wo die Markierung der Variablen, die primitive Werte beschreiben (`#` als Suffix der Variablennamen) lediglich zur besseren Lesbarkeit dienen und vom Compiler nicht berücksichtigt werden, schreibt die hier verwendete Syntax diese Markierungen ausdrücklich vor (s. Abb.3.2: *var*). Hierdurch wird eine Programmanalyse zur Bestimmung dieser Variablen vermieden. Die Markierung dieser Variablen kann bei der Transformation des vom Programmierer geschriebenen Programms in STG–Code erfolgen, da erst bei dieser Transformation die explizite Verwendung von primitiven Werten erzeugt wird. Die Markierungen werden nur vom Compiler benutzt (s. u.). Die Daten des resultierenden C–Programms besitzen keine solchen Markierungen.

3.2.5 Primitive Werte und Polymorphie

Eine Einschränkung im STG–Compiler von Peyton Jones [Pey92] verbietet die Verwendung primitiver Werte bei polymorphen Funktionen (vgl. [Pey91]). So darf z. B. der polymorphen Funktion

```
twice f x = f ( f x )
```

die eine Funktion `f` und ein Argument `x` übernimmt und `f` zweimal auf `x` anwendet, keine Funktion `f` übergeben werden, die ein Argument `x` mit primitivem Typ erwartet. Dies ist deshalb nicht möglich, da das Argument `x`, falls es einen primitiven Typ besitzt, auf einem anderen Stack an die Funktion `f` übergeben werden muß (vgl. 4.1.2 Die Argumentstacks). Eine Programmanalyse (im wesentlichen Typechecking) liefert für die Funktion `twice` jedoch nur folgendes Typschema

```
twice :: (a -> a) -> a -> a
```

(`a` stellt hier eine Typvariable dar), das keinerlei Aussage über den Typ des Arguments `x` macht. Es ist also nicht festzustellen auf welchem Stack das Argument `x` an die Funktion `f` übergeben werden muß (d. h. ob es sich um einen boxed oder einen unboxed value handelt). Abhilfe schafft hier nur die Einschränkung polymorpher Funktionen auf Zeigerobjekte (boxed values).

Erst die Markierung primitiver Variablen mit einem `#` ermöglicht die Definition polymorpher Funktionen, die primitive Werte manipulieren[4]. So ist z. B. bei der Funktion

```
twice' f x# = f ( f x# )
```

für den Compiler anhand der Markierung sofort ersichtlich, daß es sich bei `x#` um einen primitiven Wert handelt, der auf einem speziellen Stack (dem B–Stack) an die Funktion `f` übergeben werden muß.

Somit ist es möglich, polymorphe Funktionen sowohl mit boxed values als auch mit unboxed values zu verwenden. Die Funktionen müssen jeweils nur in die beiden Versionen transformiert werden. Die Verwendung der jeweils richtigen polymorphen Funktion (für boxed values oder unboxed values) liegt in der Verantwortung des Übersetzers, der STG-Code erzeugt und dabei primitive Werte überhaupt erst einführt. Die Markierung primitiver Werte liefert also mehr Informationen als ein Typinferenzalgorithmus.

[4] Die in den Veröffentlichungen von Peyton Jones verwendeten Beispielfunktionen benutzen ebenfalls ein `#` zur Markierung primitiver Variablen. Diese Markierung dient jedoch nur zur besseren Lesbarkeit und wird vom Compiler nicht berücksichtigt.

Durch die Verwendung polymorpher Funktionen in Verbindung mit primitiven Werten ist es nun auch möglich neue Datenstrukturen zu verwenden. So können nun z. B. Listen mit primitiven Werten erzeugt und manipuliert werden.

3.2.6 Primitive Listen

Listen enthalten, aufgrund polymorpher Funktionen zur Listenmanipulation (wie z. B. head – eine Funktion, die das erste Element einer beliebigen Liste liefert) nur boxed values als Elemente. Der zur Listendarstellung verwendete, zweistellige Konstruktor `Cons` enthält als erste Komponente also einen Zeiger auf eine Datenstruktur (boxed value). Diese Datenstruktur kann ein Datum (also das erste Element der Liste) enthalten oder einen Ausdruck, der bei seiner Reduktion das Listenelement liefert. Die zweite Komponente dient zur Verkettung von `Cons`–Zellen.

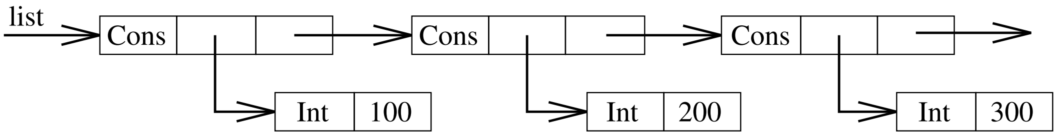


Abbildung 3.3: Struktur einer gewöhnlichen Liste

Die Fähigkeit nun auch primitive Werte mit polymorphen Funktionen zu manipulieren, gestattet die Definition von Listen, die primitive Werte als Elemente enthalten (**primitive Listen**). Die erste Komponente des Listenkonstuktors besteht hierbei nicht aus einem Zeiger auf eine Datenstruktur, die das erste Element der Liste (den primitiven Wert) enthält, sondern aus dem primitiven Wert selbst.

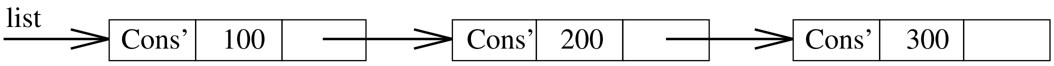


Abbildung 3.4: Struktur einer Liste mit primitiven Werten

Der Vorteil dieser primitiven Listen liegt, gegenüber einer einfachen Liste, im geringeren Platzbedarf und im schnelleren Zugriff auf die Listenelemente[5]. Allerdings können diese Listen keine unausgewerteten Ausdrücke enthalten, da unausgewertete Ausdrücke nur als boxed values darstellbar sind. Anders als bei normalen Listen muß bei der Verwendung primitiver Listen also gewährleitet sein, daß alle Ausdrücke, deren Wert in die primitive Liste eingefügt werden soll, eine Normalform besitzen, ihre Auswertung also terminiert[6]. Diese Eigenschaft kann mit Hilfe einer Terminierungsanalyse festgestellt werden (s. hierzu Panitz [Pan]).

[5]Asymptotisch betrachtet beträgt die Zugriffszeit auf die Elemente einer Liste der Länge n immer noch O(n).

[6]Normale Listen können durchaus unausgewertete Ausdrücke enthalten, die keine Normalform besitzen

Kapitel 4

Operationale Semantik

4.1 Die abstrakte Maschine

Die zugrundeliegende abstrakte Maschine entspricht der von Peyton Jones entwickelten *Spineless Tagless G–Machine* [Pey92], die eine Weiterentwicklung der *G-Machine* [Pey87] darstellt. Die Besonderheit der Spineless Tagless G–Machine ist die operationale Semantik, gegeben durch ein Zustandsübergangssystem, die direkt für die funktionale STG–Sprache angegeben wird und nicht, wie sonst üblich, für einen nicht mehr funktionalen, abstrakten Zwischencode, in den die funktionale Sprache zuvor transformiert wurde.

Die hier verwendete abstrakte Maschine enthält zwei zusätzliche Komponenten (den B–Stack und die Register), die in der Implementierung der STG–Maschine zwar Verwendung finden, in der abstrakten Maschine von Peyton Jones allerdings nicht berücksichtigt werden. Diese Komponenten dienen zur direkten Manipulation von primitiven Werten, die aus Effizienzgründen nicht im Heap dargestellt werden.

4.1.1 Komponenten der abstrakten Maschine

Die abstrakte Maschine besteht aus acht Komponenten:

1. dem Code;
2. dem Argument–Stack A, der Zeiger auf Datenstrukturen im Speicher (boxed values) enthält (dieser Stack wird in den Zustandübergangsregeln mit *as* bezeichnet);
3. dem Argument–Stack B, der primitive Werte enthält (Integer, Float, usw.; dieser Stack wird in den Zustandsübergangsregeln mit *bs* bezeichnet);
4. den Return–Registern, *regs*, die zur Übergabe von Funktionswerten dienen;
5. dem Return–Stack, *rs*, der Informationen zur Programmablaufsteuerung enthält;

6. dem Update–Stack, *us*, der Informationen über Ausdrücke (boxed values) enthält, die nach ihrer Reduktion mit ihrem Ergebnis überschrieben werden müssen;

7. dem Heap, *h*, der zur Darstellung von Funktionen und Datenobjekten dient (boxed values);

8. der globalen Umgebung, σ, die die Namen der auf oberster Ebene definierten Funktionen auf Heap–Adressen abbildet.

4.1.2 Argument–Stacks

Die Argument–Stacks dienen zur Übergabe von Argumenten bei einer Funktionsanwendung. Dabei enthält der A–Stack lediglich Adressen von Heap gehaltenen Datenstrukturen, also den boxed values, während der B–Stack primitive Werte (ganze Zahlen, Zeichen, ...) enthält. Der Einfachheit halber werden hier nur ganze Zahlen betrachtet.

4.1.3 Return–Register

Die Return–Register dienen zur Rückgabe von Funktionswerten. Sobald durch die Reduktion eines Ausdrucks ein Datenobjekt erzeugt wird, wird dieses Objekt in die Return–Register geschrieben. Daten eines primitiven Datentyps (ganze Zahlen, Zeichen, ...) werden dabei in das erste Register geschrieben, bei algebraischen Daten steht im ersten Register der Konstruktor und in den folgenden Registern dessen Argumente. Zu beachten ist, daß das Ergebnis einer Reduktion nur in den Return–Registern steht, und nicht im Heap. Das Ergebnis wird nur dann in den Heap übertragen, wenn ein zuvor reduzierter Ausdruck als 'updatable' markiert war[1].

Kann der Ausdruck aufgrund einer ungenügenden Anzahl von Argumenten nicht zu einem primitiven oder algebraischen Datum reduziert werden (partielle Applikation), so wird der Wert nicht in den Return–Registern erzeugt. Der Update–Mechanismus (s. u.) erzeugt diese partielle Applikation im Heap.

4.1.4 Return–Stack

Der Return–Stack enthält **Return–Frames**, die zur Programmablaufkontrolle dienen. Ein Return–Frame wird durch eine `case`–Anweisung erzeugt und gestattet nach der Reduktion des zu betrachtenden Ausdrucks die Verzweigung zur richtigen Alternativen der `case`–Anweisung. Hierzu besteht der Return–Frame aus den Alternativen der `case`–Anweisung und der lokalen Umgebung ρ (s. Abschnitt 4.1.9), in der die Alternativen ausgeführt werden müssen.

[1] Hierbei muß es sich nicht um den zuletzt reduzierten Ausdruck handeln

4.1.5 Update–Stack

Mit Hilfe des Update–Stacks werden Ausdrücke (d. h. ihre Repräsentation im Heap) nach ihrer Reduktion durch ihr Ergebnis ersetzt (call-by-need). Wird ein als 'updatable' markierter Ausdruck reduziert, so wird zuvor ein **Update–Frame** erzeugt, der aus vier Teilen besteht:

- den beiden Argument–Stacks (im folgenden wird mit leeren Argument–Stacks weitergerechnet),
- dem aktuellen Return–Stack (im folgenden wird auch mit einem leeren Return–Stack weitergearbeitet) und
- der Heap–Adresse des Ausdrucks, der mit seinem Wert überschrieben werden soll.

4.1.6 Heap

Der Heap bildet Heap–Adressen auf Datenstrukturen ab. Diese Datenstrukturen können beliebige boxed values, also Funktionen oder Datenobjekte darstellen und haben die Form

$$(vs \setminus\pi\ xs \texttt{ -> } e)\ ws$$

mit

- vs ist die Liste der freien Variablen (d. h. deren Namen),
- π ist das Update–Flag,
- xs ist die Liste der Argumentnamen,
- e ist ein beliebiger Ausdruck der erweiterten STG–Sprache und
- ws ist die Liste der Werte der freien Variablen.

Stellt das Heap–Objekt ein Datum dar[2], so ist die Argumentliste xs leer und die Liste ws enthält die Konstruktorkomponenten.

4.1.7 Globale Umgebung

Die globale Umgebung σ enthält Informationen über alle Funktionen, die im Programm auf oberster Ebene (global) definiert wurden. Wird eine global definierte Funktion angewandt (vgl. Regel 1), so muß, nachdem evtl. vorhandene Argumente auf die Argument–Stacks gelegt wurden, der STG–Ausdruck, der diese Funktion darstellt, reduziert werden. Dieser Ausdruck ist in einer Datenstruktur im Heap enthalten.

[2]hierbei muß es sich um einen algebraischen Datentyp handeln, also einen Konstruktor, da Daten mit primitivem Typ (unboxed values) nie im Heap dargestellt werden

Die globale Umgebung σ bildet die Namen aller global definierten Funktionen auf die Adressen ihrer Datenstrukturen im Heap ab:

$$\sigma\colon f \mapsto Addr\ a$$

4.1.8 Code

Die Code–Komponente im Zustand besteht immer aus einem der folgenden vier Befehle:

Eval e ρ wertet den Ausdruck e in der lokalen Umgebung ρ (s. 4.1.9) aus. Der Ausdruck e kann dabei beliebig verschachtelt sein.

Enter a wendet die Funktion, die im Heap unter der Adresse a dargestellt ist, auf die Argumente auf den Argument–Stacks an.

ReturnCon stellt zusammen mit dem *ReturnInt*–Befehl (s. u.) den **Return-** und den **Update-Mechanismus** der abstrakten STG-Maschine dar. Der *ReturnCon*–Befehl wird immer dann ausgeführt, wenn zuvor ein Konstruktor mit seinen Argumenten in den Return–Registern erzeugt wurde. *ReturnCon* versucht dann mit Hilfe des obersten Frames des Return–Stacks zu einer passenden Alternative, einer zuvor ausgeführten `case`–Anweisung zu verzweigen (siehe Regel 8, 9 und 10). Ein leerer Return–Stack signalisiert dabei die Notwendigkeit eines Updates (zur Erinnerung: bei der Anwendung einer als updatable gekennzeichneten Funktion wird der Return–Stack geleert).

Der bei Peyton Jones [Pey92] verwendete *ReturnCon*–Befehl enthält zusätzlich den Konstruktor und dessen Komponenten, also

$$ReturnCon\ c\ xs$$

Dies ist hier nicht notwendig, da sich der Konstruktor und seine Komponenten immer in den Return–Registern befinden.

ReturnInt wird immer dann ausgeführt, wenn zuvor ein Ergebnis mit primitivem Typ im ersten Return–Register erzeugt wurde, und verzweigt zu der passenden Alternative im obersten Element des Return–Stacks (siehe Regel 14 und 15). Hierbei kann nie ein leerer Return–Stack vorkommen, da dies zu einem Update führen würde, der eine Datenstruktur im Heap (boxed value) mit einem primitiven Wert (unboxed value) überschreibt. Primitive Werte können jedoch nicht im Heap dargestellt werden. Der Versuch ein Heap–Objekt mit einem primitiven Wert zu überschreiben stellt einen Typfehler dar, der durch den Typechecker entdeckt wird.

Auch hier wird gegenüber Peyton Jones [Pey92] ein verkürzter Befehl verwendet. Die explizite Anwendung des Befehls *ReturnInt* auf den primitiven

Wert ist nicht nötig, da dieser sich immer im ersten Return–Register befindet.

4.1.9 Lokale Umgebung ρ

Die lokale Umgebung ρ dient zur Speicherung von Informationen über lokale Funktionen und Datenobjekte. ρ bildet im Programm verwendete Namen auf Werte ab. Diese Werte können Zeiger auf Heap–Objekte sein ($Addr\ a$) oder primitive Werte ($Int\ n$). So werden z. B. die Argumente einer Funktion nach der Übergabe von den Argument–Stacks entfernt und in die lokale Umgebung ρ eingefügt. Anders als die globale Umgebung σ besitzt ρ nur eine sehr begrenzte Lebensdauer.

4.1.10 Zustand der abstrakten Maschine

Der Zustand der abstrakten Maschine wird als horizontale Reihe ihrer Komponenten dargestellt. Für die Stacks und die Register wird dabei eine Listenschreibweise verwendet. Eine Liste ist eine Folge von Elementen, die in geschweiften Klammern dargestellt wird: $\{a_1,\ a_2,\ \ldots,\ a_n\}$. Die leere Liste wird mit $\{\}$ bezeichnet. $a{:}as$ beschreibt die Liste, die mit dem Element a beginnt und $as +\!\!+\ bs$ die Liste, die mit den Elemente der Liste as beginnt. Die Variablen as und bs beschreiben dabei beliebige Listen. Die Länge einer Liste as wird mit $length\ (as)$ bezeichnet.

4.2 Zustandsübergangsregeln

Folgendes Zustandsübergangssystem beschreibt die operationale Semantik der funktionalen STG–Sprache. Es besteht aus einem Anfangszustand und einer Menge von Zustandsübergangsregeln.

Eine Zustandsübergangsregel besteht aus einem Zustand, in dem sich die abstrakte Maschine befinden muß, damit die Zustandsübergangsregel angewandt werden kann und einem Folgezustand, in dem sich die Maschine nach der Anwendung der Regel befindet. Die beiden Zustände werden durch ein $\Rightarrow$ getrennt. Komponenten des Folgezustandes, die sich gegenüber des Startzustandes der Regel nicht unterscheiden, werden durch ein '*' symbolisiert. Zur vereinfachten Darstellung werden einige Hilfsfunktionen in den Regeln verwendet.

4.2.1 Verwendete Hilfsfunktionen

In den Zustandsübergangsregeln werden die folgenden Hilfsfunktionen *val* und *split_names* verwendet.

Die Funktion *val* dient zur Bestimmung des Wertes eines im Programm verwendeten Namens und benutzt hierzu die lokale Umgebung ρ (für lokale Namen) und die globale Umgebung σ (für globale Namen):

$$\begin{array}{lllll} val & \rho & \sigma & k & = & Int\ k \\ val & \rho & \sigma & v & = & \begin{cases} \rho\ v & \text{wenn } v \in dom(\rho) \\ \sigma\ v & \text{sonst} \end{cases} \end{array}$$

Hierbei bezeichnet k ein Literal und v eine Variable. Die Funktion *val* wird ebenfalls auf Listen von Namen angewandt. In diesem Fall liefert sie eine Liste von Werten.

Die Funktion *split_names* wird bei der Anwendung einer Funktion auf Argumente benutzt (s. Regel 1), um die Argumente zur Übergabe auf den A– und den B–Stack aufzuteilen. *split_names* teilt die Namen der Argumente hierzu in zwei Namenslisten auf, von denen die erste Liste die Namen der boxed values enthält und die zweite die Namen der unboxed values. Dabei tauchen die Namen in den neuen Listen in genau derselben Reihenfolge auf, wie in der ursprünglichen Argumentliste. *split_names* benutzt zur Unterscheidung den Argumentnamen, der laut Syntax bei primitiven Werten mit `#` enden muß.

$$split\_names\ \{x_1,\ \ldots,\ x_n\} = (\{a_1,\ \ldots,\ a_k\},\ \{b_1,\ \ldots,\ b_l\})$$

$$\begin{array}{ll} \text{mit} & n = k + l \\ & a_i = x_j \Leftrightarrow x_j \text{ endet nicht mit } \texttt{\#} \\ & b_i = x_j \Leftrightarrow x_j \text{ endet mit } \texttt{\#} \\ & a_i = x_k \wedge a_j = x_l \wedge k < l \Rightarrow i < j \\ & b_i = x_k \wedge b_j = x_l \wedge k < l \Rightarrow i < j \end{array}$$

4.2.2 Anfangszustand

Ein Programm besteht aus einer Menge von Funktionsdefinitionen, von denen genau eine den Namen `main` besitzen muß:

$$\begin{array}{llllll} g_1 & \texttt{=} & \mathrm{vs}_1 & \backslash\pi_1 & \mathrm{xs}_1 & \texttt{->}\ \mathrm{e}_1 \\ \ldots \\ g_{n-1} & \texttt{=} & \mathrm{vs}_{n-1} & \backslash\pi_{n-1} & \mathrm{xs}_{n-1} & \texttt{->}\ \mathrm{e}_{n-1} \\ main & \texttt{=} & \mathrm{vs}_n & \backslash\pi_n & \mathrm{xs}_n & \texttt{->}\ \mathrm{e}_n \end{array}$$

`main` beschreibt den Ausdruck, der den Wert des Programms darstellt. Die Listen der freien Variablen sind in der Regel im Anfangszustand leer (zur genauen Bestimmung der freien Variablen siehe [Rest] und [Pey92]). Der Anfangszustand des Zustandsübergangssystems enthält diese Funktionsdefinitionen als Datenstrukturen im Heap unter den Adressen $a_1, \ldots, a_n$ und die globale Umgebung bildet die Namen dieser Funktionen auf die zugehörigen Heap–Adressen ab. Die Stacks sind am Anfang leer und begonnen wird mit der Auswertung der globalen Funktion `main`. Die leere Liste am Ende des Befehls `Eval` symbolisiert die lokale Umgebung, in der der Befehl ausgeführt werden soll. Auch diese ist zu Beginn leer.

Code	as	bs	$regs$	rs	us	h	σ
$Eval$ (main {}) {}	{}	{}	{}	{}	{}	h_{init}	σ_{init}

mit

$$\sigma_{init} = \begin{bmatrix} g_1 & \mapsto & (Addr\ a_1) \\ & \ldots & \\ g_{n-1} & \mapsto & (Addr\ a_{n-1}) \\ main & \mapsto & (Addr\ a_n) \end{bmatrix}$$

$$h_{init} = \begin{bmatrix} a_1 & \mapsto & (vs_1 \setminus \pi_1\ xs_1\ \texttt{->}\ e_1)\ (\sigma\ vs_1) \\ & \ldots & \\ a_{n-1} & \mapsto & (vs_{n-1} \setminus \pi_{n-1}\ xs_n\ \texttt{->}\ e_{n-1})\ (\sigma\ vs_{n-1}) \\ a_n & \mapsto & (vs_n \setminus \pi_n\ xs_n\ \texttt{->}\ e_n)\ (\sigma\ vs_n) \end{bmatrix}$$

Der Anfangszustand der abstrakten Maschine

Die globale Umgebung ändert sich während der gesamten Programmausführung nicht, behält also immer den Zustand σ_{init}.

4.2.3 Funktionsapplikation

Zur Reduktion einer Funktion f, angewandt auf die Argumente xs, d. h. $Eval$ (f xs), müssen zunächst die Werte der Argumente xs auf die Argument–Stacks aufgeteilt werden. Die Aufteilung erfolgt mit Hilfe der Funktion *split_names*, die die unboxed values an der Endung ihres Variablennamens (`#`) erkennt (vgl. 4.2.1). Der Funktionsname f wird von der globalen Umgebung σ bzw. der lokalen Umgebung ρ auf einen boxed value im Heap abgebildet, der die Funktion f darstellt. Der zugehörige STG–Ausdruck wird nun mit Hilfe des Befehls *Enter* auf die Argumente auf den Stacks angewandt.

	$Eval\ (f\ xs)\ \rho$	as	bs	$regs$	rs	us	h	σ
		mit $val\ \rho\ \sigma\ f = Addr\ a$						
$\Rightarrow$	$Enter\ a$	$as'' ++ as$	$bs'' ++ bs$	$*$	$*$	$*$	$*$	$*$
		mit $(as',\ bs') = split\_names\ xs$ $as'' = val\ \rho\ \sigma\ as'$ $bs'' = val\ \rho\ \sigma\ bs'$						

Regel 1: Auswertung einer Funktionsanwendung

Der Befehl *Enter a* wendet nun die Funktion, die im Heap an der Adresse a steht, auf die Argumente auf dem A– und B–Stack an. Dabei sind drei Fälle zu beachten:

1. Die Funktion ist als 'updatable' gekennzeichnet. In diesem Fall erwartet die Funktion keine Argumente[3].
2. Die Funktion ist als 'nonupdatable' gekennzeichnet und wird auf eine ihrer Stelligkeit entsprechenden Anzahl von Argumenten angewandt.
3. Die Funktion ist als 'nonupdatable' gekennzeichnet, die Anzahl der auf den Argument–Stacks vorhandenen Argumente ist jedoch kleiner als die Funktionsstelligkeit.

Im ersten Fall, in dem keine Argumente vorhanden sind, wird lediglich der im Heap stehende Ausdruck ausgewertet (*Eval e*). Die lokale Umgebung ρ, bildet infolge nicht vorhandener Argumente lediglich die Namen der freien Variablen auf deren Werte ab (vgl. Regel 2. Vor dieser Reduktion wird jedoch ein Update–Frame (s. Kapitel 4.1.5) auf den Update–Stack gelegt, um den Ausdruck (bzw. dessen Repräsentation im Heap) nach seiner Reduktion durch sein Ergebis zu ersetzen. Bei der Erzeugung des Update–Frames wird außer der Heapadresse a des Ausdrucks auch der Return– und die Argument–Stacks im Update–Frame gesichert. Im weiteren wird nun mit einen leeren Return–Stack und leeren Argument–Stacks fortgefahren. Diese leeren Stacks zeigen die Notwendigkeit eines Updates an:

- Wird der Ausdruck e zu einem Datum (Konstruktor) reduziert, so wird mit Hilfe des *ReturnCon*–Befehls versucht, eine passende Alternative einer

[3]Eine Funktion, die Argumente zu ihrer Auswertung benötigt, ist bereits in Head–Normal–Form. Eine Reduktion dieser Funktion (ohne Argumente) würde zu keinem neuen Ergebnis führen, ein Update wäre also sinnlos. Daraus folgt, daß nur Funktionen, die keine Argumente erwarten als 'updatable' gekennzeichnet sein können (nicht müssen).

vorangegangenen *case*–Anweisung im Return–Stack zu finden (nur eine *case*–Anweisung führt zur Auswertung eines Ausdrucks). Ist der Return–Stack bei der Ausführung des *ReturnCon*–Befehls leer, so wird ein Update ausgeführt. (s. Regel 8).

- Kann der Ausdrucks e aufgrund nicht vorhandener Argumente nicht (oder nicht weiter) reduziert werden (partielle Applikation), so wird ein Update ausgeführt (s. Regel 4), d. h. ein boxed value im Heap wird mit diesem (so weit reduzierten) Ausdruck überschrieben.

$$
\begin{array}{llllllll}
 & \mathit{Enter}\ a & as & bs & regs & rs & us & h \quad \sigma \\
 & & & & & \multicolumn{3}{l}{h\ a = (vs\ \texttt{\textbackslash u}\ \{\}\ \texttt{->}\ e\ )\ ws_f} \\
\Rightarrow & \mathit{Eval}\ e\ \rho & \{\} & \{\} & * & \{\} & (as,\ bs,\ rs,\ a){:}us & * \quad * \\
 & & & & & \multicolumn{3}{l}{\text{mit } \rho = [vs \mapsto ws_f]}
\end{array}
$$

Regel 2: Anwendung einer später zu überschreibenden Funktion

Im zweiten Fall (die Funktion ist als 'nonupdatable' gekennzeichnet und wird auf eine ihrer Stelligkeit entsprechenden Anzahl von Argumenten angewandt) wird lediglich der Ausdruck e reduziert, wobei die benötigten Argumente von den Stacks genommen und in die lokale Umgebung eingefügt werden. Alle anderen Komponenten der Maschine bleiben unverändert.

$$
\begin{array}{lllllllll}
 & \mathit{Enter}\ a & as & bs & regs & rs & us & h[a \mapsto (vs\ \texttt{\textbackslash n}\ xs\ \texttt{->}\ e\ )\ ws_f] & \sigma \\
 & & & & \text{mit} & \multicolumn{4}{l}{(xs_a,\ xs_b) = \mathit{split\_names}\ xs} \\
 & & & & & \multicolumn{4}{l}{\mathit{length}(as) \geq \mathit{length}(xs_a)} \\
 & & & & & \multicolumn{4}{l}{\mathit{length}(bs) \geq \mathit{length}(xs_b)} \\
\Rightarrow & \mathit{Eval}\ e\ \rho & as' & bs' & * & * & * & * & * \\
 & & & & \text{mit} & \multicolumn{4}{l}{\rho = [vs \mapsto ws_f,\ xs_a \mapsto ws_a,\ xs_b \mapsto ws_b]} \\
 & & & & & \multicolumn{4}{l}{(xs_a,\ xs_b) = \mathit{split\_names}\ xs} \\
 & & & & & \multicolumn{4}{l}{ws_a \mathbin{+\!\!+} as' = as} \\
 & & & & & \multicolumn{4}{l}{ws_b \mathbin{+\!\!+} bs' = bs} \\
 & & & & & \multicolumn{4}{l}{\mathit{length}(ws_a) = \mathit{length}(xs_a)} \\
 & & & & & \multicolumn{4}{l}{\mathit{length}(ws_b) = \mathit{length}(xs_b)}
\end{array}
$$

Regel 3: Anwendung einer Funktion auf eine ausreichende Anzahl von Argumenten

Im dritten Fall, in dem nicht genügend Argumente zur Reduktion des Ausdrucks

(i) $$(vs\ \texttt{\textbackslash n}\ xs\ \texttt{->}\ e)\ ws_f$$

der im Heap an der Adresse a steht, vorhanden sind, entsteht eine partielle Applikation. Die zur Reduktion fehlenden Argumente wurden durch eine früher reduzierte Funktion von den Argument–Stacks entfernt und im Update–Stack gesichert, um anzuzeigen, daß diese Funktion mit der entstandenen partiellen Applikation überschrieben werden soll. Diese früher reduzierte, zu überschreibende Funktion wird nun durch die partielle Applikation ersetzt. Dabei werden die bereits auf den Argument–Stacks vorhandenen Argumente (as und bs) zu freien Variablen der partiellen Applikation, die gegenüber (i) eine um diese Argumente verringerte Stelligkeit ($length(xs_2) = length(xs)$ - ($length$ (as) + length (bs))) besitzt:

(ii) $$(vs \mathbin{+\!\!+} xs_1\ \texttt{\textbackslash n}\ xs_2\ \texttt{->}\ e)\ (ws_f \mathbin{+\!\!+} as \mathbin{+\!\!+} bs)$$

Hierbei sind xs_1 beliebige, verschiedene Variablennamen, mit denen der Ausdruck e auf die neu entstandenen, freien Variablen zugreifen kann ($length(xs_1) = length$ (as) + $length$ (bs))). Mit xs_2 werden die noch nicht vorhandenen Argumente bezeichnet. Diese Darstellung bedingt jedoch die erneute Übersetzung des Ausdrucks e, da dieser im neu entstandenen Heap–Objekt (ii) auf einen Teil seiner ursprünglichen Argumente (und zwar die, die jetzt freie Variablen sind) auf eine andere Art zugreifen muß. Dies würde eine Übersetzung zur Laufzeit erfordern, die nicht möglich ist, da der Compiler keinen Maschinencode, sondern C–Code erzeugt. Stattdessen wird ein Heap–Objekt (iii) erzeugt, das bei seiner Reduktion die neu entstandenen, freien Variablen zurück auf die Argument–Stacks legt und dann das ursprüngliche Heap–Objekt (i) darauf anwendet. Hierzu muß auch die Adresse a des Heap–Objektes (i) in das neue Heap–Objekt (iii) aufgenommen werden:

(iii) $$(f{:}xs_1\ \texttt{\textbackslash n}\ \{\}\ \texttt{->}\ f\ xs_1)\ (a{:}as \mathbin{+\!\!+} bs)$$

Auch hier sind die xs_1 beliebige, verschiedene Variablennamen, die die neu entstandenen freien Variablen bezeichnen. Das ursprüngliche Heap–Objekt a, das nun auch eine freie Variable darstellt, wird mit f bezeichnet. Beim Aufruf der Funktion f (also der ursprünglichen Funktion (i)), befinden sich die freien Variablen wieder auf den Argument–Stacks, womit der ursprünglich für e erzeugte Code verwendet werden kann.

Folgende Zustandsübergangsregel beschreibt diesen Update mit einer partiellen Applikation. Hierbei werden die für die neu entstandenen, freien Variablen verwendeten Namen in ys_a und ys_b aufgeteilt, da die Variablennamen für 'unboxed values' (also Argumente vom B–Stack) mit einem `#` enden müssen[4].

Nachdem die im Update–Frame enthaltenen Argument–Stacks und der Return–Stack wiederhergestellt sind, wird erneut versucht, das Heap–Objekt an der Adresse a zu reduzieren.

$$
\begin{array}{llllllll}
\mathit{Enter}\ a & as & bs & regs & \{\} & (as_u,\ bs_u,\ rs_u,\ a_u){:}us & h & \sigma
\end{array}
$$

$$
\text{mit}\quad \begin{array}{l}
h\ a = (vs\ \texttt{\textbackslash n}\ xs\ \texttt{->}\ e\ )\ ws_f \\
length\ (as) < length\ (xs_a) \vee length\ (bs) < length\ (xs_b) \\
(xs_a,\ xs_b) = split\_names\ xs
\end{array}
$$

$$\Rightarrow$$

$$
\begin{array}{llllllll}
\mathit{Enter}\ a & as{+\!\!+}as_u & bs{+\!\!+}bs_u & * & rs_u & us & h_u & *
\end{array}
$$

$$
\text{mit}\quad \begin{array}{l}
h_u = h[a_u \mapsto ((f{:}ys_a{+\!\!+}ys_b)\ \texttt{\textbackslash n}\ \{\}\ \texttt{->}\ f\ (ys_a{+\!\!+}ys_b))\ a{:}as{+\!\!+}bs] \\
\text{wobei } f,\ ys_a \text{ und } ys_b \text{ beliebige, verschiedene Variablen} \\
\text{sind mit} \\
length(ys_a) = length(as) \wedge length(ys_b) = length(bs)
\end{array}
$$

Regel 4: Anwendung einer Funktion auf eine nicht ausreichende Anzahl von Argumenten

4.2.4 Der `let(rec)`–Ausdruck

Der `let`– bzw. `letrec`–Ausdruck ist der einzige STG–Ausdruck, der Datenstrukturen im Heap erzeugen kann. Diese Strukturen können beliebig komplexe STG–Funktionen darstellen. Die nach dem Schlüsselwort `let` bzw. `letrec` definierten Funktionen $x_1, \ldots, x_n$ werden im Heap erzeugt. Die in diesen Definitionen verwendeten freien Variablen dürfen bei einer `let`–Anweisung nur aus der lokalen Umgebung ρ stammen. Bei einer `letrec`–Anweisung dürfen zusätzlich die definierten Funktionen $x_1, \ldots, x_n$ verwendet werden (also die Umgebung ρ'). Anschließend wird der Ausdruck e in der Umgebung ρ', also in der um die Definitionen erweiterten Umgebung ρ, ausgewertet (Regel 5).

4.2.5 Der `case`–Ausdruck

Der `case`–Ausdruck der Form

$$\texttt{case}\ e\ \texttt{of}\ alts$$

wertet den Ausdruck e aus und verzweigt dann in Abhängigkeit des Ergebnisses von e zu einer der in *alts* definierten Alternativen. Diese Alternativen müssen in

[4] dies ist notwendig, um festzustellen, auf welchen der beiden Argument–Stacks eine freie Variable gelegt werden muß

$$
\begin{array}{ll}
\mathit{Eval} \left[\begin{array}{l} \texttt{let(rec)} \\ x_1 = vs_1 \ \backslash\pi_1 \ xs_1 \ \texttt{->} \ e_1 \\ \ldots \\ x_n = vs_n \ \backslash\pi_n \ xs_n \ \texttt{->} \ e_n \\ \texttt{in} \ e \end{array} \right] & \rho \quad as \quad bs \quad regs \quad rs \quad us \quad h \quad \sigma \\
\Rightarrow & \\
\mathit{Eval} \ e \ \rho' & \quad * \quad * \quad * \quad * \quad * \quad h' \quad *
\end{array}
$$

$$
\begin{array}{lll}
\text{mit} \quad \rho' & = & \rho[x_1 \mapsto \mathit{Addr}\ a_1, \ \ldots, x_n \mapsto \mathit{Addr}\ a_n] \\
h' & = & h \left[\begin{array}{l} a_1 \mapsto (vs_1 \ \backslash\pi_1 \ xs_1 \ \texttt{->} \ e_1) \ (\rho_{rhs} \ vs_1) \\ \ldots \\ a_n \mapsto (vs_n \ \backslash\pi_n \ xs_n \ \texttt{->} \ e_n) \ (\rho_{rhs} \ vs_n) \end{array} \right] \\
\rho_{rhs} & = & \left\{ \begin{array}{ll} \rho & \text{für } \texttt{let} \\ \rho' & \text{für } \texttt{letrec} \end{array} \right.
\end{array}
$$

Regel 5: Auswertung eines `let(rec)`–Ausdrucks

der gleichen Umgebung ρ ausgewertet werden, wie der Ausdruck e selbst. Da e ein beliebig komplexer Ausdruck sein kann und die lokale Umgebung ρ verändern kann, muß ρ zusammen mit den Alternativen *alts* zwischengespeichert werden. Hierzu dient der Return–Stack, der während der Reduktion von e die Alternativen und die Umgebung ρ als Return–Frame enthält. Nach der Auswertung von e wird mit Hilfe des *ReturnCon*– bzw. *ReturnInt*–Befehls der Return–Frame wieder vom Stack entfernt, die darin enthaltene lokale Umgebung wieder hergestellt und zu der entsprechenden Alternativen verzweigt (s. Regel 8 und 14).

$$
\begin{array}{llllllll}
 & \mathit{Eval}\ (\texttt{case}\ e\ \texttt{of}\ \mathit{alts})\ \rho & as & bs & regs & rs & us & h \quad \sigma \\
\Rightarrow & \mathit{Eval}\ e\ \rho & * & * & * & (\mathit{alts},\ \rho){:}rs & * & * \quad *
\end{array}
$$

Regel 6: Auswertung eines `case`–Ausdrucks

4.2.6 Konstruktoren

Bei der Auswertung von Konstruktoren werden zum ersten Mal die Return–Register verwendet. Diese Register dienen zur Übergabe von Funktionswerten. Konstruktoren werden dabei wie folgt übergeben: Das Konstruktortag steht im ersten Return–Register und die Komponenten des Konstruktors (falls vorhanden) stehen in den folgenden Registern. Diese Methode zur Rückgabe von Funktionswerten wird bereits in der Implementierung der STG–Maschine von Peyton Jones [Pey92] verwendet. Das dort beschriebene Zustandsübergangssystem enthält diese Register jedoch nicht. Durch die Erweiterung des Systems wird die Fähigkeit zur Manipulation dieser Register bereits in der abstrakten Maschine zur Verfügung gestellt[5].

Ein durch eine Reduktion entstehender Konstruktor c wird also in den Return–Registern erzeugt. Die Speicherung dieses Konstruktors in einem Heap–Objekt erfolgt nur durch einen (evtl.) auftretenden Update (s. Regel 11). Wird kein Update vorgenommen, so geht der Konstuktor bei der nächsten Verwendung der Register verloren.

	Eval $(c\ xs)$	ρ	as	bs	$regs$	rs	us	h	σ
$\Rightarrow$	*ReturnCon*		$*$	$*$	$c{:}xs$	$*$	$*$	$*$	$*$

Regel 7: Auswertung eines Konstruktors

Der Befehl *ReturnCon* steuert nun in Abhängigkeit des in den Registern stehenden Konstruktors die Fortführung des Programms (Return– und Update–Mechanismus), indem er versucht eine passende Alternative eines vorangegangenen *case*–Ausdrucks, der zur Auswertung des Konstruktors geführt hat, zu finden. Dabei sind drei Fälle zu beachten:

1. Der oberste Frame des Return–Stacks enthält eine zum Konstruktor passende Alternative.
2. Der oberste Frame des Return–Stacks enthält keine zum Konstruktor passende, aber eine Default–Alternative.
3. Der Return–Stack ist leer.

Bei einem vorhandenen Return–Frame ohne eine passende Alternative und ohne `default`–Alternative (d. h. es ist keine Regel anwendbar) hält die Maschine. Dies entspricht einem Laufzeitfehler des Programms.

Im ersten Fall, in dem eine konstruktorspezifische Alternative im obersten Return–Frame enthalten ist, wird diese Alternative reduziert, wobei die ebenfalls im Return–Frame enthaltene lokale Umgebung ρ benutzt wird. ρ wird dabei um die nun zur Verfügung stehenden Konstruktorkomponenten erweitert (d. h. ρ bildet die in der Alternativen enthaltenen Komponentennamen vs auf die in den Registern stehenden Werte xs ab). Die Register selbst werden dabei nicht verändert. Gleichzeitig wird der oberste Return–Frame vom Return–Stack entfernt.

[5] Ziel ist, diese Register bei der Erzeugung von Maschinencode auf Hardwareregister abzubilden. Bei der Erzeugung von C–Code werden diese Register durch globale Variablen dargestellt. Auch ohne die Abbildung dieser globalen Variablen auf Hardwareregister, wie dies beim GNU–C–Compiler möglich ist, ergeben sich gegenüber der Rückgabe von Funktionswerten in Heap–Objekten einige Vorteile (s. 5.2.1)

$$\begin{array}{lcccccccc}
\mathit{ReturnCon} & as & bs & c{:}xs & \left(\begin{array}{l}\ldots \\ c\ vs\ \texttt{->}\ e\ ,\rho \\ \ldots\end{array}\right){:}rs & us & h & \sigma \\
\Rightarrow & & & & & & & \\
\mathit{Eval}\ e\ \rho[vs \mapsto xs] & * & * & * & rs & * & * & *
\end{array}$$

Regel 8: Verzweigung zu einer algebraischen Alternativen

Ist keine zum Konstruktor passende Alternative vorhanden, so wird (falls vorhanden) die Default–Alternative ausgewertet. Es gibt zwei Arten von algebraischen Default–Alternativen:

- Default–Alternativen, die mit dem Schlüsselwort `default` eingeleitet werden.
- Default–Alternativen, die mit einer Variablen eingeleitet werden.

Wird bei der Default–Alternative das Schlüsselwort `default` verwendet, geht der reduzierte Konstruktor verloren. Die Alternative wird in der lokalen Umgebung ρ reduziert, die nicht erweitert werden muß. Auch hier wird der oberste Return–Frame entfernt.

$$\begin{array}{lcccccccc}
\mathit{ReturnCon} & as & bs & c{:}xs & \left(\begin{array}{l}c_1\ vs_1\ \texttt{->}\ e_1 \\ \ldots \\ c_m\ vs_m\ \texttt{->}\ e_m \\ \texttt{default}\ \texttt{->}\ e_d\end{array} ,\rho\right){:}rs & us & h & \sigma \\
\text{mit} \quad c \neq c_i\ (1 \leq i \leq m) & & & & & & & \\
\Rightarrow & & & & & & & \\
\mathit{Eval}\ e_d\ \rho & * & * & * & rs & * & * & *
\end{array}$$

Regel 9: Verzweigung zu einer algebraischen `default`–Anweisung ohne Variable

Bei der Verwendung einer Default–Variablen v muß der gesamte Konstruktor (also einschließlich seiner Komponenten) an diese Variable v gebunden werden. Hierzu wird eine Datenstruktur im Heap erzeugt, die den Konstruktor aufnimmt. Die in dieser Heapstruktur als freie Variablen verwendeten beliebigen, aber verschiedenen Variablennamen vs (s. Regel 10 erlauben dabei den Zugriff auf die Konstruktorkomponenten. Die Default–Alternative wird dann in der lokalen Umgebung ρ reduziert, die um die Variable v erweitert wurde. v stellt dabei die Adresse der neuen Heapstruktur dar.

$$
\begin{array}{llllllllll}
\mathit{ReturnCon} & \mathit{as} & \mathit{bs} & c{:}\mathit{xs} & \left(\begin{array}{l} c_1\ \mathit{vs}_1\ \texttt{->}\ e_1 \\ \ldots \\ c_m\ \mathit{vs}_m\ \texttt{->}\ e_m \\ v\ \texttt{->}\ e_d \end{array}, \rho\right){:}\mathit{rs} & \mathit{us} & h & \sigma \\
\multicolumn{4}{l}{\text{mit}\quad c \neq c_i \quad (1 \leq i \leq m)} \\
\\
\Rightarrow \\
\mathit{Eval}\ e_d\ \rho[v \mapsto a] & * & * & * & \mathit{rs} & * & h' & * \\
\\
 & \multicolumn{7}{l}{\text{mit}\quad h' = h[a \mapsto (\mathit{vs}\ \texttt{\textbackslash n}\ \{\}\ \texttt{->}\ c\ \mathit{vs})\ \mathit{xs}]} \\
 & \multicolumn{7}{l}{\qquad\ \ \mathit{vs}\ \text{sind beliebige, verschiedene Variablen mit}} \\
 & \multicolumn{7}{l}{\qquad\ \ \mathit{length}(\mathit{vs}) = \mathit{length}(\mathit{xs})}
\end{array}
$$

Regel 10: Verzweigung zu einer algebraischen `default`–Anweisung mit Variable

Falls beim Auftreten des *ReturnCon*–Befehls der Return–Stack leer ist, so muß zuvor eine Funktion mit dem Update–Flag `\u` angewandt worden sein, die den Return-Stack im Update–Stack gesichert hat, um die Notwendigkeit eines Updates anzuzeigen. Die Heapstruktur, die diese Funktion darstellt und deren Adresse a_u ebenfalls im Update–Stack enthalten ist, muß nun mit dem Ergebnis der Reduktion, also dem Konstruktor, überschrieben werden[6]. Danach wird mit dem Befehl *ReturnCon* und dem wieder hergestellten Return–Stack ein weiteres Mal versucht eine passende Alternative zu finden.

$$
\begin{array}{lllllllll}
\mathit{ReturnCon} & \{\} & \{\} & c{:}\mathit{xs} & \{\} & (\mathit{as}_u,\ \mathit{bs}_u,\ \mathit{rs}_u,\ a_u){:}\mathit{us} & h & \sigma \\
\\
\Rightarrow \\
\mathit{ReturnCon} & \mathit{as}_u & \mathit{bs}_u & * & \mathit{rs}_u & \mathit{us} & h_u & * \\
\\
 & \multicolumn{7}{l}{\text{mit}\quad h_u = h[a_u \mapsto (\mathit{vs}\ \texttt{\textbackslash n}\ \{\}\ \texttt{->}\ c\ \mathit{vs}\ )\ \mathit{xs}]} \\
 & \multicolumn{7}{l}{\qquad\ \ \mathit{vs}\ \text{sind beliebige, verschiedene Variablen mit}} \\
 & \multicolumn{7}{l}{\qquad\ \ \mathit{length}(\mathit{vs}) = \mathit{length}(\mathit{xs})}
\end{array}
$$

Regel 11: Update eines Heap–Objektes mit einem Konstruktor

4.2.7 Primitive Werte

Auch bei der Auswertung primitiver Werte (Literale oder Variablen) werden die Return–Register verwendet. Hierzu wird das erste Return–Register mit dem

[6]In der abstrakten Maschine wird hierzu einfach die Heapstruktur an der Adresse a_u durch das neue Heap–Objekt ersetzt. In der Implementierung treten hier einige Probleme auf. So kann z. B. die alte Heapstruktur zu klein sein, um das neue Ergebnis aufzunehmen (vgl. 7.2).

primitiven Wert geladen.

Bei der Auswertung eines Literals k wird einfach das erste Return–Register mit dem Wert k geladen. Alle anderen Komponenten der Maschine bleiben unverändert.

$Eval\ k\ \rho$	as	bs	$regs$	rs	us	h	σ
$\Rightarrow$							
$ReturnInt$	$*$	$*$	$\{k\}$	$*$	$*$	$*$	$*$

Regel 12: Auswertung eines Literals

Bei der Auswertung einer Variablen f, die von der lokalen Umgebung ρ auf einen primitiven Wert k abgebildet wird, wird dieser Wert k in das erste Return–Register geschrieben.

$Eval\ (f\ \{\})\ \rho$	as	bs	$regs$	rs	us	h	σ
	mit $\rho\ f = Int\ k$						
$\Rightarrow$							
$ReturnInt$	$*$	$*$	$\{k\}$	$*$	$*$	$*$	$*$

Regel 13: Auswertung einer Variablen, die ein Literal bezeichnet

Der Befehl *ReturnInt* wählt nun genauso wie *ReturnCon* aus dem obersten Element des Return–Stacks die richtige Alternative aus. Im Gegensatz zum *ReturnCon*–Befehl gibt es hier allerdings nur zwei mögliche Fälle:

1. Der Return–Frame enthält eine, dem Wert im ersten Return–Register entsprechende Alternative

2. Der Return–Frame enthält keine dem Wert entsprechende Alternative, aber eine Default–Alternative

Der beim *ReturnCon*–Befehl vorhandene dritte Fall, in dem der Return–Stack leer (d. h. im Update–Frame gesichert) ist, kann hier nicht auftreten, da ein leerer Return–Stack einen nötigen Update mit dem Ergebnis des gerade reduzierten Ausdrucks anzeigt. Das Überschreiben eines Heap–Objektes mit einem primitiven Wert ist jedoch nicht möglich (vgl. 4.1.6).

Beim Vorhandensein einer passenden Alternative im Return–Frame, wird diese mit dem Befehl *Eval* in der Umgebung ρ ausgewertet. Der oberste Return–Frame wird dabei vom Return–Stack entfernt.

ReturnInt	*as*	*bs*	$\{k\}$	$(\ldots,\, k$ `->` $e, \ldots,\, \rho)$:*rs*	*us*	*h*	σ
$\Rightarrow$							
Eval $e\ \rho$	*	*	*	*rs*	*	*	*

Regel 14: Auswahl einer primitiven Alternativen

Ansonsten wird die Default-Alternative ausgewertet. Die lokale Umgebung ρ wird dabei um die Default-Variable v erweitert, an die der Wert k im ersten Return–Register gebunden wird. Dann wird der zugehörige Ausdruck e_d in dieser erweiterten lokalen Umgebung reduziert.

ReturnInt	*as*	*bs*	$\{k\}$	$\left(\begin{array}{l} k_1 \texttt{ -> } e_1 \\ \ldots \\ k_n \texttt{ -> } e_n \\ v \texttt{ -> } e_d \end{array},\rho\right)$:*rs*	*us*	*h*	σ
mit $k \neq k_i \quad (1 \leq i \leq n)$							
$\Rightarrow$							
Eval $e_d\ \rho[v \mapsto Int\ k]$	*	*	*	*rs*	*	*	*

Regel 15: Auswahl einer primitiven Default–Alternativen mit Variablen

4.2.8 Arithmetische Operationen

Arithmetische Operationen, wie z. B. Addition, Multiplikation usw., die mit primitiven Werten arbeiten, schreiben ihr Ergebnis (einen primitiven Wert, in Regel 16 als $i_1 \oplus i_2$ dargestellt) nach der Berechnung ebenfalls in das erste Return–Register. Somit kann auch hier mit dem Befehl *ReturnInt* fortgefahren werden.

Eval $(\oplus\ \{x_1,\ x_2\})\ \rho$	*as*	*bs*	*regs*	*rs*	*us*	*h*	σ
mit $\rho\ x_1 = Int\ i_1$							
$\rho\ x_2 = Int\ i_2$							
$\Rightarrow$							
ReturnInt	*	*	$\{i_1 \oplus i_2\}$	*	*	*	*

Regel 16: Anwendung eines arithmetischen Operators

Tritt eine arithmetische Operation in einem `case`–Ausdruck auf,

$$\texttt{case}\ \oplus\ \{x_1,\ x_2\}\ \texttt{of}\ alts$$

so wird zunächst der Ausdruck $\oplus\ \{x_1,\ x_2\}$ reduziert (Regel 6), dessen Ergebnis in das erste Return–Register geschrieben wird (Regel 16). Dann wird mit dem Befehl *ReturnInt* fortgefahren, der zur entsprechenden Alternative des `case`–Ausdrucks verzweigt (Regeln 14 bzw. 15).

Die Auswertung eines solchen `case`–Ausdrucks kann optimiert werden, indem direkt nach der Auswertung der arithmetischen Operation zur entsprechenden Alternative verzweigt wird. Damit wird die Erzeugung eines Return–Frames gespart. Diese Optimierung führt zu folgenden beiden Regeln:

$$\begin{array}{lllllllll}
\mathit{Eval} \left(\begin{array}{l} \texttt{case} \oplus \{x_1, x_2\} \text{ of} \\ k_1 \texttt{ -> } e_1 \\ \ldots \\ k_n \texttt{ -> } e_n \\ v \texttt{ -> } e_d \end{array}\right) & \rho & as & bs & regs & rs & us & h & \sigma \\
\text{mit} \quad \rho\ x_1 = \mathit{Int}\ i_1 & & & & & & & & \\
\phantom{\text{mit}} \quad \rho\ x_2 = \mathit{Int}\ i_2 & & & & & & & & \\
\Rightarrow & & & & & & & & \\
\mathit{Eval}\ e_j\ \rho & & * & * & * & * & * & * & * \\
\text{falls } i_1 \oplus i_2 = k_j \text{ für } j \in \{\mathit{1} \ldots n\} & & & & & & & &
\end{array}$$

Regel 17: Optimierte Auswertung eines `case`–Ausdrucks

$$\begin{array}{lllllllll}
\mathit{Eval} \left(\begin{array}{l} \texttt{case} \oplus \{x_1, x_2\} \text{ of} \\ k_1 \texttt{ -> } e_1 \\ \ldots \\ k_n \texttt{ -> } e_n \\ v \texttt{ -> } e_d \end{array}\right) & \rho & as & bs & regs & rs & us & h & \sigma \\
\text{mit} \quad \rho\ x_1 = \mathit{Int}\ i_1 & & & & & & & & \\
\phantom{\text{mit}} \quad \rho\ x_2 = \mathit{Int}\ i_2 & & & & & & & & \\
\Rightarrow & & & & & & & & \\
\mathit{Eval}\ e_d\ \rho[v \mapsto i_1 \oplus i_2] & & * & * & * & * & * & * & * \\
\text{falls } i_1 \oplus i_2 \neq k_j \text{ für } (\mathit{1} \leq j \leq n) & & & & & & & &
\end{array}$$

Regel 18: Optimierte Auswertung eines `case`–Ausdrucks (Default-Alternative)

4.2.9 Beispiel

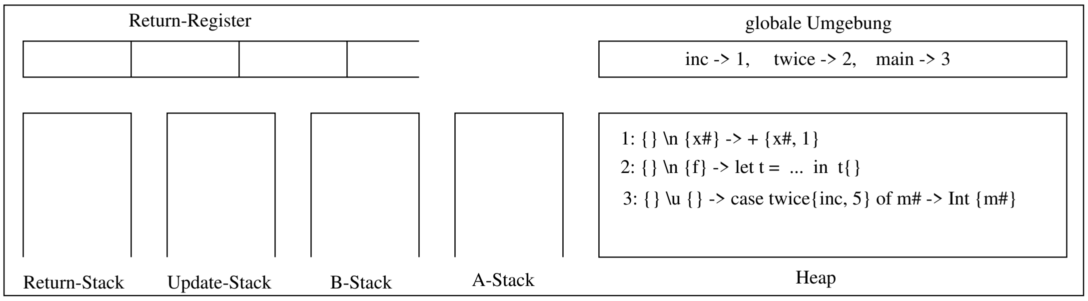
Auf den nächsten Seiten werden die Zustandsübergänge für das folgende Beispielprogramm dargestellt.

```
inc = {} \n {x#} -> + {x#, 1}

twice = {} \n {f} -> let t = {f} \n {x#} -> case f {x#} of
                                                n# -> f {n#}
                     in t {}

main = {} \u {} -> case twice {inc, 5} of
                     m# -> Int {m#}
```

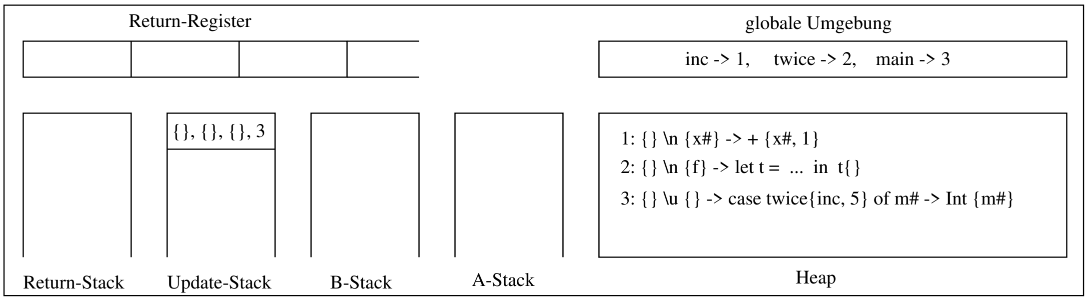
Zur besseren Lesbarkeit wird der verwendete Konstruktor als `Int` dargestellt. Der Zustand der abstrakten Maschine wird nicht nach jedem STG–Befehl angegeben, sondern nur, wenn sich der Zustand verändert hat.

Nach der Ausführung des Programms befindet sich das Ergebnis (`Int 7`) in den Rückgaberegistern.

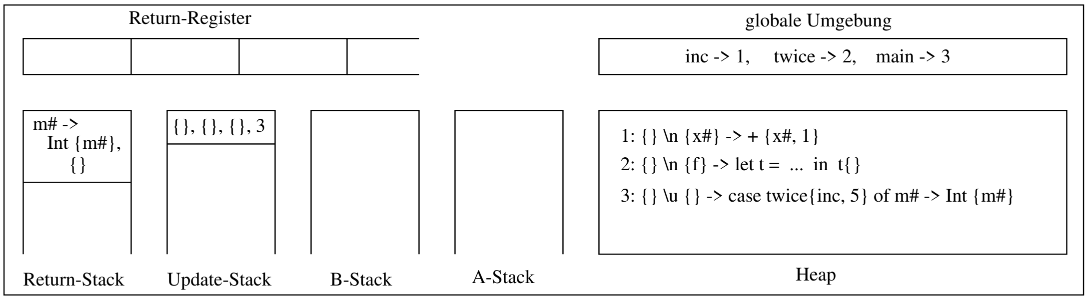
Der Anfangszustand



Eval (main { }) { }
Enter 3



Eval (case twice {inc, 5} of m# -> Int {m#}) { }



Eval (twice {inc, 5}) { }

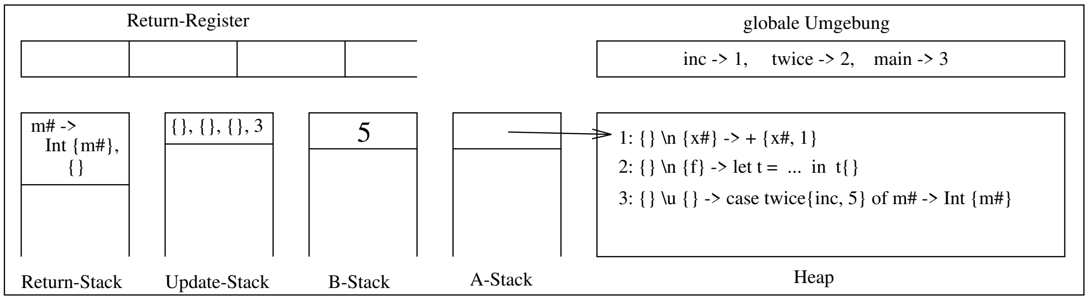
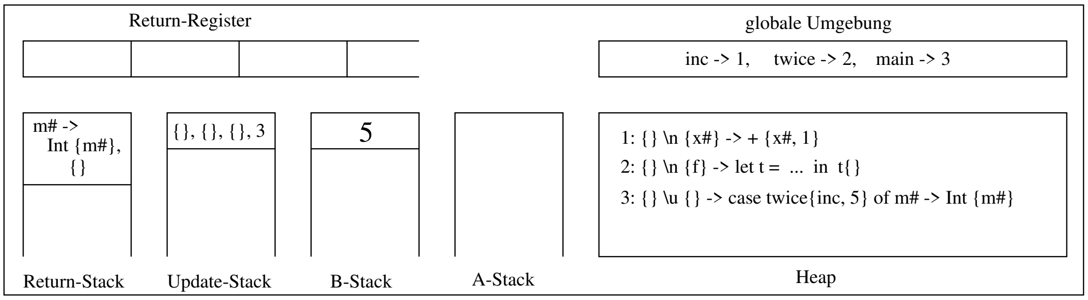
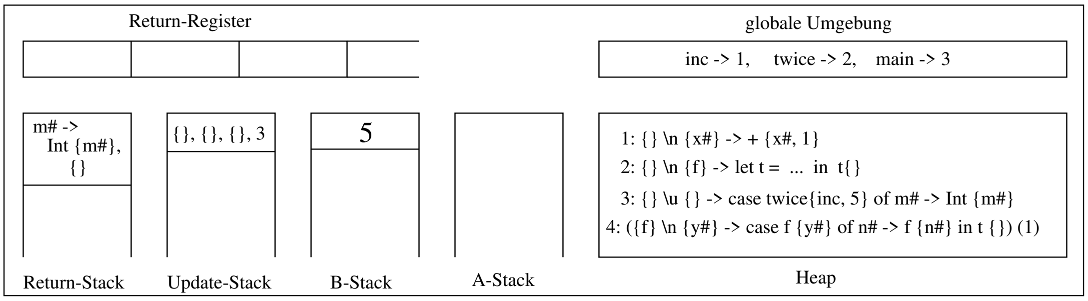


Abbildung 4.1: *Zustandsübergänge eines Beispielprogramms*

Enter 2

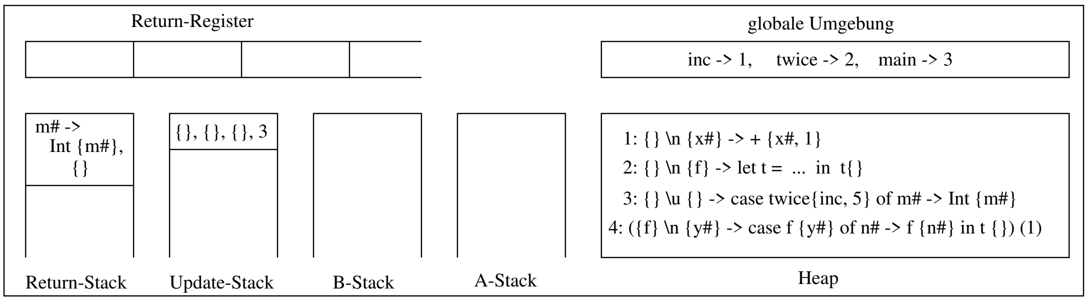


Eval (let t = {f} \n {y#} -> case f {y#} of n# -> f {n#} in t {}) { f -> 1}



Eval (t {}) { f -> 1}

Enter 4



Eval (case f {y#} of n# -> f {n#}) {f -> 1, y# -> 5}

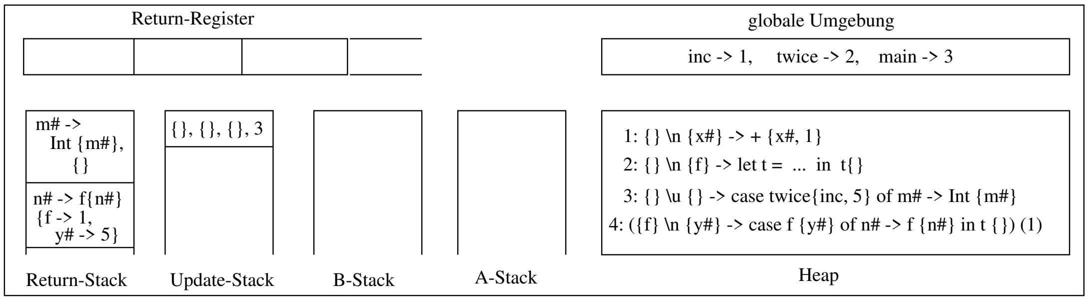
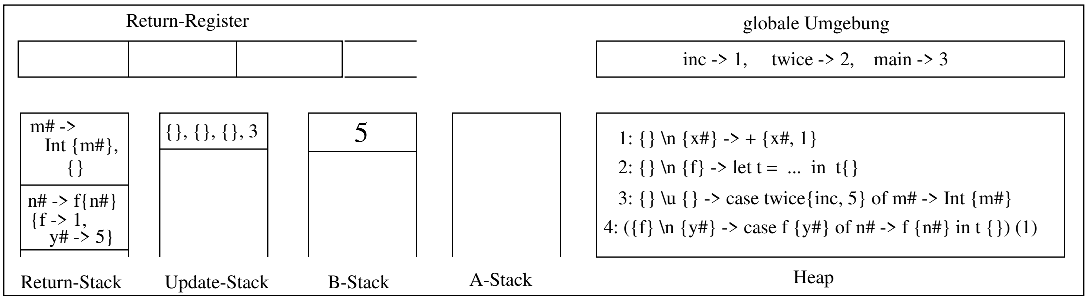
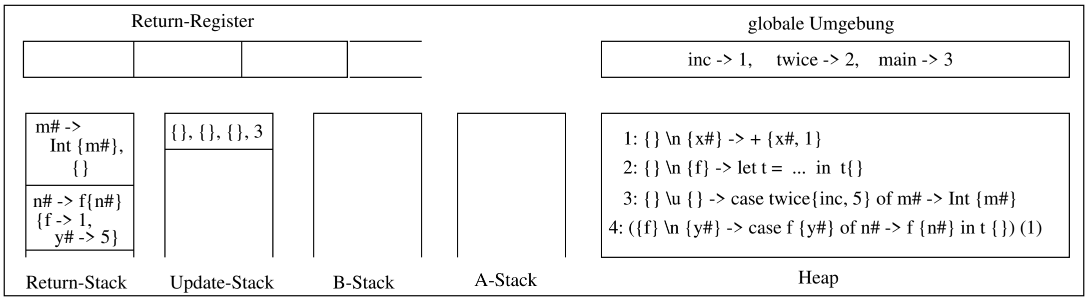


Abbildung 4.2: *Zustandsübergänge eines Beispielprogramms (Fortsetzung)*

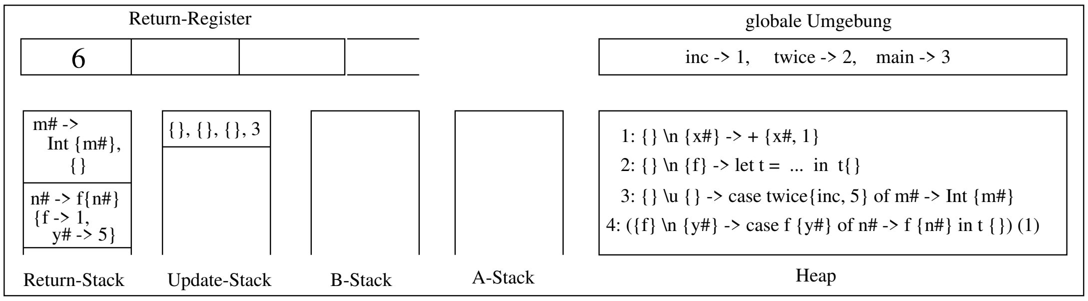
Eval (f{y#}) {f -> 1, y# -> 5}



Enter 1



Eval (+ {x#, 1}) {x# -> 5}



ReturnInt

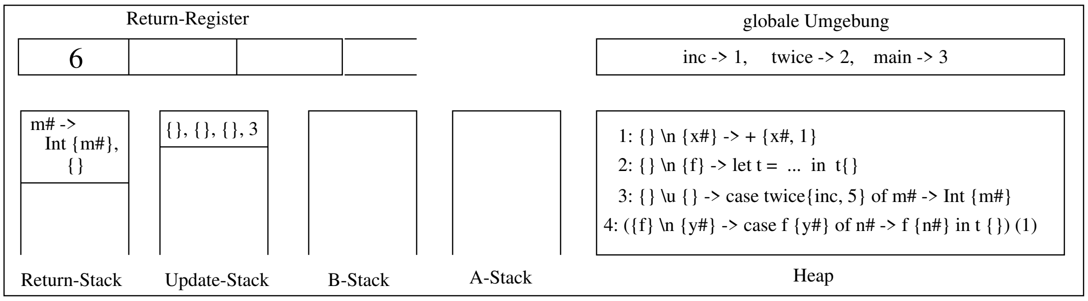
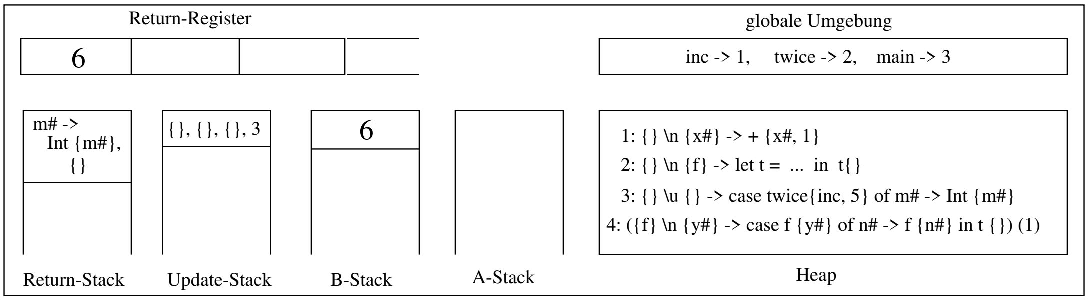
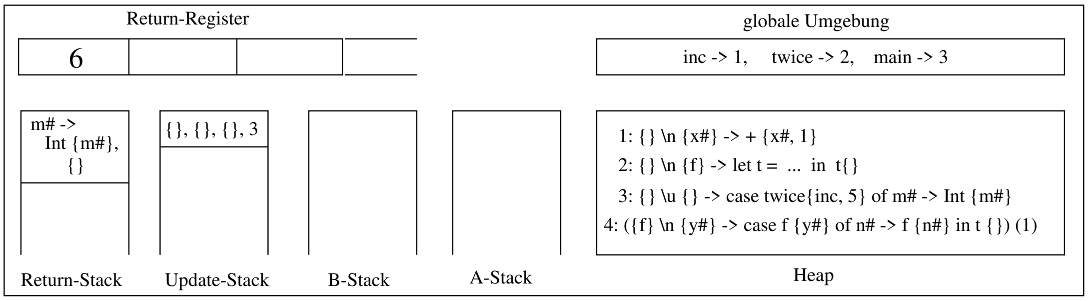


Abbildung 4.3: *Zustandsübergänge eines Beispielprogramms (Fortsetzung)*

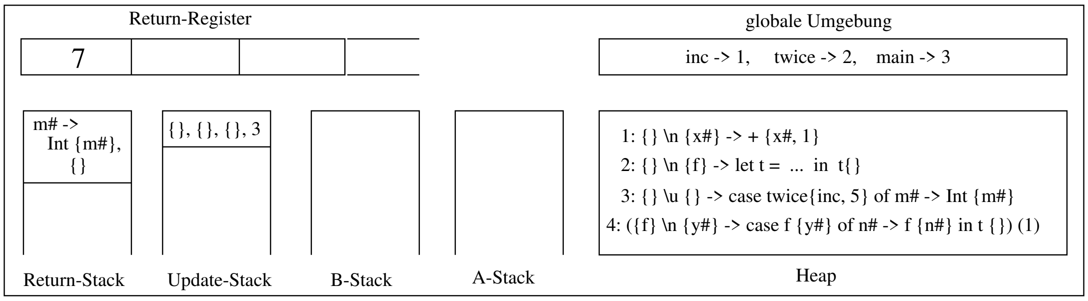
Eval (f{n#}) {f -> 1, y# -> 5, n# -> 6}



Enter 1



Eval (+ {x#, 1}) {x# -> 6}



ReturnInt

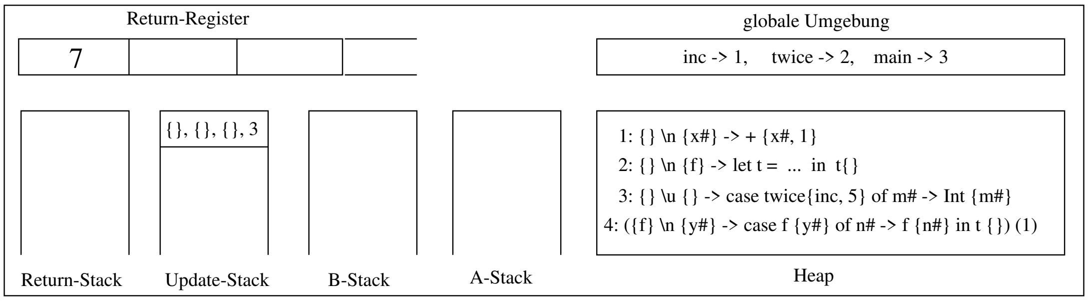
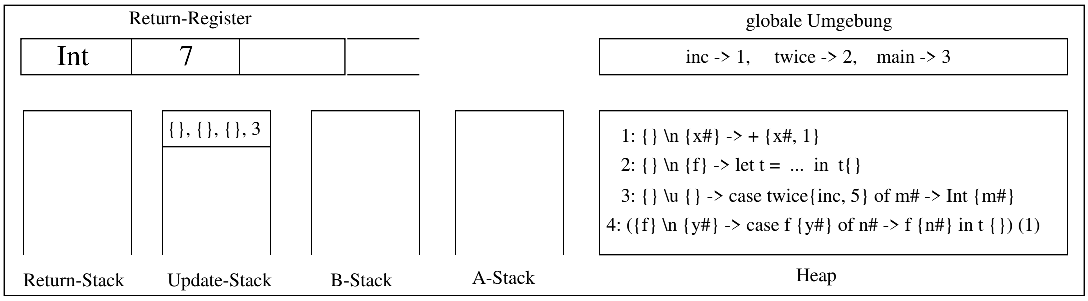


Abbildung 4.4: *Zustandsübergänge eines Beispielprogramms (Fortsetzung)*

Eval (Int {m#}) {n# -> 7}



ReturnCon

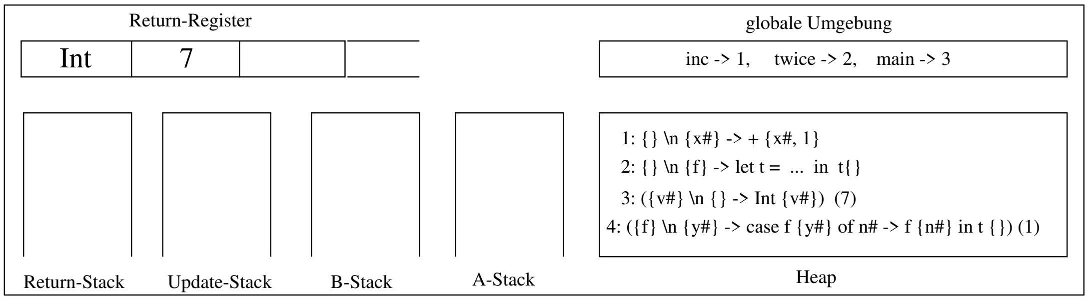


Abbildung 4.5: *Zustandsübergänge eines Beispielprogramms (Fortsetzung)*

Kapitel 5

Darstellung der abstrakten Maschine in C

In den folgenden Abschnitten wird die Darstellung der Komponenten der abstrakten Maschine in C–Code beschrieben. Hierbei werden (insbesondere in Abschnitt 5.6.2) die Implementierungen anderer funktionaler Übersetzer (Clean und Haskell) betrachtet und Vor- und Nachteile dieser Implementierungen diskutiert. In Fällen, in denen es sich nicht ohne weiteres entscheiden läßt, ob eine „Implementierungsvariante" effizienteren Code liefert, als eine andere, werden Laufzeitvergleiche durchgeführt. Für diese Vergleiche werden funktionale Beispielprogramme verwendet, die so gewählt wurden, daß sich jeweils die Vor- und Nachteile einer Implementierungsvariante zur Laufzeit bemerkbar machen.

5.1 Die Argument–Stacks

Wie aus der operationalen Semantik hervorgeht, werden zwei Stacks zur Argumentübergabe verwendet. Diese beiden Stacks werden mit A und B bezeichnet.

Der A–Stack dient ausschließlich der Übergabe von Zeigern auf Heapknoten (boxed values). Der B–Stack hingegen wird, außer zur Übergabe von primitiven Werten (unboxed values), auch zur Speicherung von Return– und Update–Frames verwendet (s. Abschnitte 5.3 und 5.4). Diese Aufteilung in A– und B–Stack erfolgt, um die Garbage–Collection (s. Abschnitt 7.3) zu vereinfachen. Diese muß lediglich alle Elemente des A–Stacks verfolgen[1]. Es ist also nicht (wie bei der Verwendung eines einzelnen Stacks) nötig zu unterscheiden, ob es sich bei einem Stack–Element um einen Zeiger oder einen primitiven Wert handelt.

Zur effizienten Platzausnutzung wachsen die beiden Stacks einander entgegen (Abb. 5.1), sodaß ein Programm nie aufgrund eines vollen Stacks abgebrochen werden muß, während im anderen Stack noch Speicherplatz vorhanden ist.

[1] Tatsächlich müssen auch die Update–Frames im B–Stack von der Garbage–Collection betrachtet werden, da diese Zeiger auf die zu überschreibenden Heapknoten enthalten

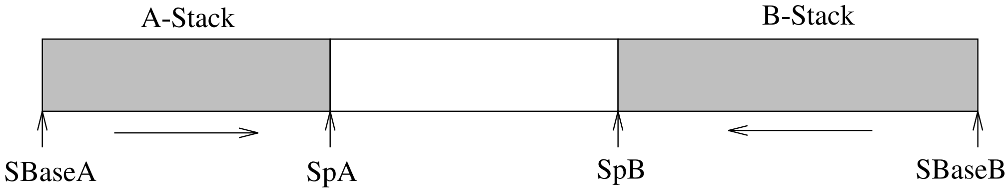


Tabelle 5.1: *Die Argument–Stacks in der Implementierung wachsen zur effizienten Speicherausnutzung einander entgegen*

Die Argument–Stacks sind mit jeweils zwei Zeigern ausgestattet. Ein Zeiger (`SpA` bzw. `SpB`) zeigt auf den nächsten freien Speicherplatz im Stack (also über das oberste Stackelement). Der andere Zeiger (`SBaseA` bzw. `SBaseB`) verweist auf das unterste Element im Stack. Bei einem leeren Stack verweisen beide Zeiger auf den ersten freien Speicherplatz. Die Anzahl der auf einem Stack vorhandenen Argumente kann also aus der Differenz der beiden Stack–Zeiger ermittelt werden.

5.1.1 Die Sicherung der Argument–Stacks

Wird während der Programmausführung eine Funktion angewandt, die später mit ihrem Ergebnis überschrieben werden soll (also das Update–Flag `\u` für updatable besitzt), so müssen die Argument–Stacks A und B in einem Update–Frame gespeichert werden. Im folgenden wird dann mit leeren Argument–Stacks weitergerechnet (vgl. Die operationale Semantik: Regel 2).

Die Sicherung der Stacks erfolgt, indem die Basiszeiger der Stacks (`SBaseA` bzw. `SBaseB`) auf den Wert von `SpA` bzw. `SpB` gesetzt werden. Die Differenz der beiden Zeiger ist dann also 0, d. h. es kann mit leeren Argument–Stacks weitergerechnet werden. Der urspüngliche Wert der Basiszeiger wird dabei in einem Update–Frame (vgl. Abschnitt 5.4.1) zwischengespeichert, um die Stacks später wieder herstellen zu können.

5.1.2 Die Abbildung des B–Stacks auf den System–Stack

Bei der Erzeugung von C–Funktionen aus STG–Funktionen liegt eine Abbildung der Argumente der STG–Funktionen auf Argumente der C–Funktionen nahe. Diese Argumente würden dann auf dem System–Stack übergeben, wodurch eine explizite Manipulation des B–Stacks entfallen würde.

Diese Abbildung scheitert bei der Übergabe von Zeigern auf Heapknoten (also Argumenten auf dem A–Stack) allerdings an der Notwendigkeit, diese Zeiger bei einer Garbage–Collection zu verfolgen. Eine Manipulation des System–Stacks in C ist zwar möglich, führt jedoch zu einem stark maschinenabhängigen Code. Eine solche Manipulation sollte (wenn überhaupt) nur dann durchgeführt werden, falls Assembler–Code erzeugt wird.

Die Übergabe primitiver Werte auf dem System–Stack erscheint unproblematisch, da diese Argumente bei einer Garbage–Collection nicht verfolgt werden müssen. Auch die Bestimmung der Anzahl der übergebenen Argumente ist möglich (z. B. durch ein zusätzliches Argument, das diese Anzahl angibt). Bei der Manipulation freier Variablen ergibt sich jedoch ein entscheidender Nachteil:

Sei f eine n–stellige Funktion höherer Ordnung, deren Wert eine m–stellige Funktion ist. Wird nun f auf $n + m$ Argumente angewandt, so kann f reduziert werden. Das Resultat, d. h. die m–stellige Funktion muß nun auf die letzten m Argumente (bzw. freien Variablen), die an die Funktion f übergeben wurden, angewandt werden. Werden zur Reduktion der Funktion f andere Funktionen angewandt, die dann die m–stellige Funktion liefern, so müssen jeder dieser Funktionen die letzten m Argumente von f übergeben werden. Dies bedeutet jedoch, daß diese Argumente mehrfach auf dem System–Stack abgelegt werden. Auf einem gesonderten B–Stack hingegen sind diese Argumente nur einmal vorhanden, wodurch Speicherplatz und Laufzeit gespart werden.

5.2 Die Return–Register

Zur Rückgabe von Funktionswerten werden Return–Register verwendet. Diese Register sind in der Lage einen primitiven Wert (unboxed value) oder einen Zeiger auf einen boxed value aufzunehmen. Sobald eine Funktion bei ihrer Reduktion ein Datum, d. h. einen primitiven Wert (unboxed value) oder einen Konstruktor (evtl. mit Komponenten) erzeugt, wird dieses Datum in die Return–Register geschrieben. Für primitive Werte genügt hierzu ein Register, während für Konstruktoren, je nach Stelligkeit mehrere Register benötigt werden. Dabei wird das erste Register mit dem Konstruktortag beschrieben und die folgenden mit jeweils einer Komponente des Konstruktors.

Das Ziel der expliziten Verwendung dieser Return–Register ist die Erzeugung von effizientem Code, in dem diese Register auf Hardwareregister abgebildet werden. Diese Abbildung stellt bei der Erzeugung von Assemblercode kein Problem dar. Bei der Erzeugung von C–Code müssen jedoch compiler–spezifische Fähigkeiten in Anspruch genommen werden, um diese Abbildung zu realisieren. Die Verwendung der in C vorhandenen Speicherklasse „register“ (vgl. Kernighan, Ritchie [KR83]) ist nicht möglich, da diese sich nur auf lokale Variablen einer Funktion anwenden läßt, während der Rückgabewert einer Funktion außerhalb der Funktion, also global zur Verfügung stehen muß. Eine spezielle Fähigkeit des GNU–C–Compiler ist die Abbildung globaler Variablen auf Hardwareregister, die für die Implementierung der Return–Register verwendet wird.

Die Register der abstrakten Maschine werden also durch globale C–Variablen mit den Namen `RetReg1`, `RetReg2`, ... dargestellt und werden (für eine konkrete

Maschine) auf Hardwareregister abgebildet.

5.2.1 Die Anzahl der verwendeten Return–Register

Zur Rückgabe eines primitiven Wertes genügt ein einzelnes Return–Register. Bei der Rückgabe eines n–stelligen Konstruktors werden jedoch (n + 1) Register benötigt (ein Register für das Konstruktor–Tag und n für die Komponenten)[2]. Durch eine einfache Analyse des zu übersetzenden STG–Programms läßt sich der Konstruktor mit der höchsten Stelligkeit ermitteln. Diese maximale Stelligkeit ist in einem STG–Programm zwar fest, kann aber, durch benutzerdefinierte Datentypen, beliebig groß sein. Deshalb wird (insbesondere bei der Erzeugung von Assemblercode) für Konstruktoren deren Größe (d. h. deren Stelligkeit + 1) die Anzahl der vorhandenen Hardwareregister übersteigt eine Sonderbehandlung nötig.

Peyton Jones [Pey92] erzeugt aus diesen Konstruktoren einen Heapknoten. Die Funktion liefert dann einen Zeiger auf diesen Heapknoten zurück. Der Zugriff auf das Konstruktortag und alle Komponenten erfolgt dann indirekt über diesen Zeiger. Bei der Erzeugung von C–Code, bei der die Return–Register durch globale

[2] In der Implementierung werden tatsächlich (n + 2) Register verwendet. Das zusätzliche Register enthält die Anzahl der verwendeten Register.

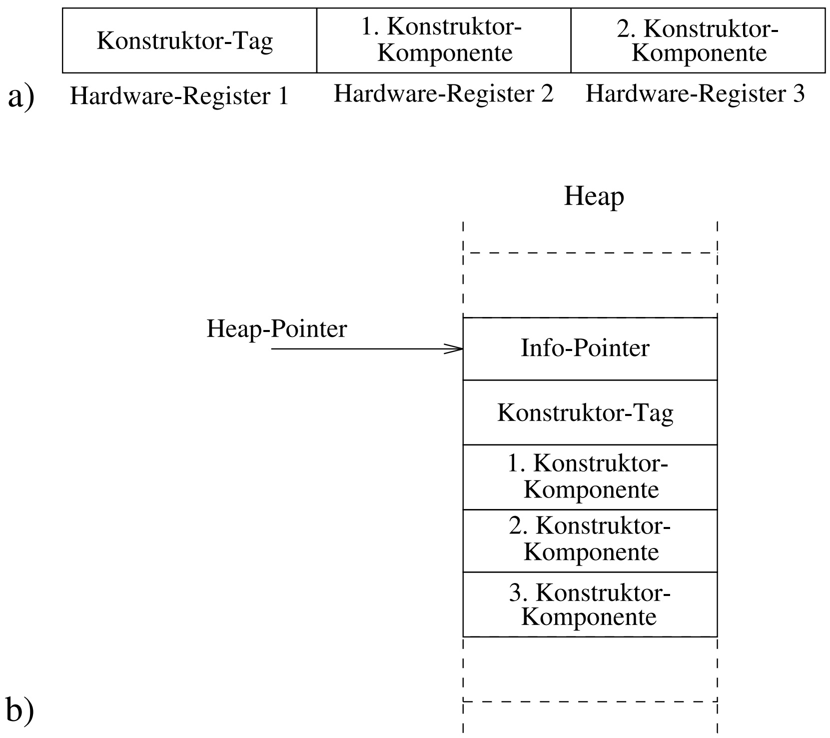


Abbildung 5.1: *Rückgabe eines Konstruktors bei Peyton Jones: a) Der Konstruktor ist klein genug, um ihn in den Hardwareregistern zu übergeben. b) Die Anzahl der Hardwareregister ist zu klein, um den Konstruktor aufzunehmen. Ein komplettes Heap-Objekt muß erzeugt werden.*

C–Variablen dargestellt und dann auf Hardwareregister abgebildet werden, bietet sich jedoch eine effizientere Lösung an:

Da die Anzahl der in einem C–Programm verwendeten globalen Variablen beliebig ist, werden gerade soviele verwendet, daß der größte, im STG–Programm verwendete Konstruktor darin Platz findet. Von diesen globalen C–Variablen werden dann soviele wie möglich auf Hardwareregister abgebildet. Wird nun ein Konstruktor erzeugt, dessen Größe die Anzahl der Hardwareregister übersteigt, so wird dieser trotzdem in den Return–Registern (d. h. in den globalen C–Variablen) erzeugt. Hierdurch erfolgt der Zugriff auf diese Konstruktoren, zumindest für das Konstruktortag und die ersten Komponenten über Hardwareregister. Die Komponenten in den globalen C–Variablen, die nicht mehr auf Hardwareregister abgebildet werden können, werden wie gewöhnliche globale C–Variablen durch einen direkten Speicherzugriff erreicht.

Konstruktor-Tag	1. Konstruktor-Komponente	2. Konstruktor-Komponente	3. Konstruktor-Komponente
1. Return-Register = Hardware-Register	2. Return-Register = Hardware-Register	3. Return-Register = Hardware-Register	4. Return-Register = globale C-Variable

Abbildung 5.2: *Effizientere Rückgabe eines Konstruktors: Reicht die Anzahl der Hardwareregister nicht aus, um alle Konstruktorkomponenten aufzunehmen, so werden die restlichen Komponenten in globalen C–Variablen gehalten.*

Dies verringert die Zugriffszeit auf Konstruktoren, deren Größe die Anzahl der Hardwareregister übersteigt, erheblich. Außerdem hat diese Darstellung positive Auswirkungen auf den Update–Mechanismus.

5.2.2 Verkleinerte Update–Frames

Zur Programmablaufsteuerung werden Update–Frames verwendet. Diese werden auf dem Update–Stack erzeugt, um nach der Reduktion eines Ausdrucks einen Heapknoten mit dem Ergebnis der Reduktion zu überschreiben (vgl. Regel 2).

In der Implementierung enthalten diese Frames die Adressen der Funktionen, mit denen das Programm fortgesetzt werden muß (also die Adresse der Update–Funktion, die den Heapknoten überschreibt — s. Abschnitt 5.4).

Soll nun ein Heapknoten durch einen Konstruktor mit seinen Komponenten überschrieben werden, so muß der Update–Funktion bekannt sein, wo sich der Konstruktor befindet (im Heap oder in den Return–Registern). In der Implementierung von Peyton Jones [Pey92], wird dies erreicht, indem der Update–Frame eine Sprungtabelle[3] enthält. Diese Sprungtabelle enthält für jeden möglichen Konstruktor einen Zeiger auf die entsprechende Update–Funktion. Für jeden Typ im Programm, der in einen Update verwickelt ist, muß eine solche Tabelle existieren.

Werden Konstruktoren, wie hier, immer in den Return–Registern erzeugt, so kann auf eine Sprungtabelle verzichtet werden, da nur eine Update–Funktion benötigt wird, die bei allen Konstruktoren Verwendung findet. Ein Update–Frame enthält also nur einen Zeiger auf die einzige Update–Funktion (zum genauen Aufbau des Update–Frames s. Abschnitt 5.4.1).

5.3 Der Return–Stack

Der Return–Stack dient zur Aufnahme von Return–Frames, die die Verzweigung zur richtigen Alternativen einer `case`–Anweisung gestatten (*Pattern Matching*).

[3] Diese Tabelle wird allerdings zusammen mit einem Return–Frame abgelegt (vgl. Abschnitt 5.5)

Allerdings wird der Return–Stack im C–Programm nicht durch einen eigenen Stack realisiert. Vielmehr werden die Return–Frames auf dem B–Stack verwaltet. Der B–Stack repräsentiert also sowohl den Argument–Stack für primitive Argumente, als auch den Return–Stack[4].

5.3.1 Die Return–Frames

Die Return–Frames bestehen aus ein oder zwei Zeigern. Ein Zeiger verweist auf eine Funktion, die die richtige Alternative eines `case`–Ausdrucks auswählt (**Standard–Return–Funktion**) und einen Teil des Rückgabemechanismuses darstellt (s. Abschnitt 7.1), der andere Zeiger verweist auf eine Sprungtabelle, die die Adressen der C–Funktionen, die die Alternativen darstellen, enthält (Abb. 5.5a, Seite 53).

Existiert in einem `case`–Ausdruck nur eine mögliche Alternative, so enthält der Return–Frame nicht einen Zeiger auf die Standard–Return–Funktion, sondern auf den Code dieser Alternative[5]. Der Zeiger auf die Sprungtabelle kann in diesem Fall entfallen.

Handelt es sich bei dem Ergebnis des im `case`–Ausdruck zu reduzierenden Ausdrucks um einen primitiven Wert, so kann keine Sprungtabelle verwendet werden. In diesem Fall wird das Pattern Matching durch *if–else* Ketten realisiert (vgl. Abschnitt 6.4.3). Auch in diesem Fall besteht der Return–Frame nur aus einem Zeiger, der auf eine C–Funktion verweist, die das Pattern Matching vornimmt.

5.3.2 Die Notwendigkeit eines Funktionszeigers im Return–Frame

Der Aufruf der Standard–Return–Funktion nach der Erzeugung eines Konstruktors kann auch direkt erfolgen (ohne deren Adresse im Return–Frame abzulegen). Der Return–Frame besteht dann nur aus dem Zeiger auf die Sprungtabelle. Allerdings muß dann auch eine einzelne Alternative über die Sprungtabelle (also indirekt) angesprungen werden. Der eigentliche Grund für die Verwendung des Funktionszeigers ist jedoch die notwendige Unterscheidung zwischen Return– und Update–Frames. Update–Frames werden ebenfalls auf dem B–Stack abgelegt. Bei der Erzeugung eines Konstruktors ist nicht klar, ob es sich bei dem Frame auf dem B–Stack um einen Update– oder einen Return–Frame handelt. Um nicht zur Laufzeit eine Markierung zu testen, enthalten beide Frames einen Zeiger auf die entsprechende Funktion (Einsprungpunkt), die einfach angesprungen wird. Peyton Jones [Pey92] verwendet diese Funktionszeiger nicht. Dies hat zur Folge,

[4] Zusätzlich werden noch Update–Frames im B–Stack gespeichert (vgl. Abschnitt 5.4).

[5] Diese Alternative muß dann überprüfen, ob die Auswahl korrekt war, da evtl. eine zum Ergebnis der Reduktion passende Alternative im Programm nicht angegeben ist.

daß auch einzelne Alternativen über eine Sprungtabelle angesprungen werden müssen (S. Abschnitt 5.5).

5.4 Der Update–Stack

Der Update–Stack enthält Frames, die dazu dienen, Heapknoten nach ihrer Reduktion mit ihrem Ergebnis zu überschreiben. Genau wie der Return–Stack wird auch der Update–Stack nicht durch einen gesonderten Stack repräsentiert. Der B–Stack dient auch hier zur Speicherung der Update–Frames.

5.4.1 Die Update–Frames

Der Semantik zufolge enthält ein Update–Frame die beiden Argument–Stacks, den Return–Stack und den Zeiger auf den Heapknoten, der mit seinem Ergebnis überschrieben werden soll (vgl. Abschnitt 4.1.5).

Die Sicherung der Argument–Stacks erfolgt, indem die Basiszeiger der Stacks, die auf das unterste Element des jeweiligen Stacks verweisen, verändert werden. Indem diese Zeiger genau wie die Zeiger `SpA` und `SpB` auf den ersten freien Speicherplatz im Stack zeigen, erscheinen die Stacks im weiteren als leer. Zur Wiederherstellung der Stacks müssen im Update–Frame lediglich die ursprünglichen Werte der Basiszeiger gespeichert werden.

Eine Sicherung des Return–Stacks ist nicht nötig. Zwar wird laut Semantik ein Update nur dann ausgeführt, falls der Return–Stack leer ist. Durch die gemeinsame Verwendung des B–Stacks sowohl für Update– als auch für Return–Frames, wird jedoch ein vorhandener Return–Frame durch die Erzeugung eines Update–Frames „abgedeckt“.

Die Update–Frames bestehen also aus den beiden Basiszeigern der Argument–Stacks und dem Zeiger auf den zu überschreibenden Heapknoten. Zusätzlich enthält jeder Update–Frame einen Zeiger auf die **Standard–Update–Funktion**. Diese Funktion führt die Ersetzung des entsprechenden Heapknoten durch (s. Abschnitt 7.2). Die Standard–Update–Funktion ist für alle Update–Frames identisch. Trotzdem ist diese Funktion in jedem Update–Frame enthalten, um diesen Frame von einem Return–Frame zu unterscheiden. Return–Frames enthalten ebenfalls einen Zeiger auf eine Funktion. Somit muß während der Programmausführung nicht getestet werden, ob der oberste Frame auf dem B–Stack ein Update– oder Return–Frame ist. Es genügt, die im Frame enthaltene Funktion anzuspringen.

Ein Funktionszeiger in den beiden Frames ist nicht die einzige Möglichkeit Return– von Update–Frames zu unterscheiden. In der Tat kommt Peyton Jones [Pey92] ohne diese Zeiger aus. Dies ist jedoch mit einem erheblich höheren Speicherbedarf verbunden und bringt keine Vorteile in der Laufzeit, wie folgender Abschnitt zeigt.

5.5 Pattern Matching und Updates

Nach der Erzeugung eines Konstruktors muß entweder ein Return oder ein Update durchgeführt werden, je nachdem welcher Frame auf dem B–Stack liegt. Da dies jedoch bei der Erzeugung des Konstruktors nicht bekannt ist, muß ein einheitliches Verfahren zur Behandlung von Return– und Update–Frames benutzt werden. Hierzu enthalten die hier verwendeten Return– und Update–Frames jeweils einen Zeiger auf eine Funktion, die unabhängig vom Typ des Frames, angesprungen wird (vgl. Abschnitte 5.3.1 und 5.4.1). Im STG–Compiler von Peyton Jones [Pey92] wird auf einen solchen Funktionszeiger verzichtet. Dies führt zu einem relativ komplexen Update– und Return–Mechanismus:

Der Return–Frame bei Peyton Jones [Pey92] enthält lediglich einen Zeiger auf die Sprungtabelle mit den Adressen der einzelnen Alternativen. Nach der Erzeugung eines Konstruktors wird also die richtige Alternative über diesen Zeiger angesprungen. (Im Gegensatz dazu wird hier zunächst die Standard–Return–Funktion angesprungen, die dann mit Hilfe der Sprungtabelle zur richtigen Alternativen verzweigt). Dies bedeutet jedoch, daß auch eine einzelne Alternative im Return–Frame als Tabelle dargestellt werden muß. Diese Alternative wird also indirekt über den Tabellenzeiger angesprungen. (Bei den hier verwendeten Return–Frames kann der Funktionszeiger direkt auf die Alternative verweisen, der Sprung erfolgt also ohne den indirekten Zugriff über den Tabellenzeiger).

Damit die Update–Frames genauso benutzt werden können, müssen auch diese aus einer Sprungtabelle bestehen. Es wird also für jeden möglichen Konstruktor ein Eintrag benötigt. Die hier verwendeten Update–Frames enthalten lediglich einen Zeiger auf die Standard–Update–Funktion. Peyton Jones [Pey92] benötigt tatsächlich eine Sprungtabelle, da er verschiedene Update–Funktionen benötigt, je nachdem, ob ein Konstruktor im Heap oder in den Registern erzeugt wurde (vgl. Abschnitt 5.2). Diese Sprungtabelle kann aber nicht immer bei der Übersetzung einer als updatable gekennzeichneten STG–Funktion erzeugt werden, da es sich um eine polymorphe Funktion handeln kann, deren Typ also nicht bekannt ist. Deshalb erzeugt Peyton Jones [Pey92] diese Sprungtabelle zusammen mit dem Return–Frame. Die Sprungtabelle im Return–Frame enthält jetzt für jeden möglichen Konstruktor zwei Einträge: Die Adresse der Alternative, die bei der Erzeugung des Konstruktors angesprungen werden soll und die Update–Funktion, die einen Heapknoten mit diesem Konstruktor überschreibt (Abb. 5.5b). Der über diesem Return–Frame liegende Update–Frame enthält nun lediglich eine Standard–Funktion, die im darunterliegenden Return–Frame die dem Konstruktor entsprechende Update–Funktion aufruft. Diese Update–Funktion führt dann den Update durch und springt anschließend zur entsprechenden Return–Funktion im gleichen Return–Frame. Da bei der Erzeugung des Return–Frames jedoch nicht bekannt ist, ob ein Update folgt, muß jeder Return–Frame diese erweiterte Sprungtabelle enthalten. Noch komplizierter wird dieser Update–Mechanismus, wenn der Return–Frame von mehreren Update–Frames abgedeckt wird.

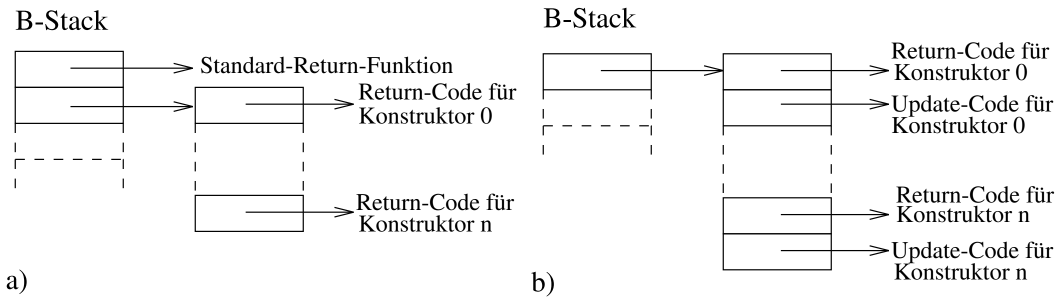


Tabelle 5.2: *Die Return–Frames: a) der hier verwendete Return–Frame; b) der bei Peyton Jones verwendete Return–Frame*

Ein Vergleich der beiden Verfahren zeigt einen deutlichen Nachteil bei der Ausführung eines Updates bei Peyton Jones [Pey92]. Während im hier verwendeten Verfahren die Adresse der Update–Funktion direkt dem Update–Frame entnommen werden kann, erfolgt der Zugriff auf die Adresse der Standard–Update–Funktion bei Peyton Jones indirekt über den Tabellenzeiger im Update–Frame. Diese Funktion führt jedoch nicht den Update aus, sondern ruft nur die eigentliche Update–Funktion auf. Deren Adresse befindet sich in der Sprungtabelle des Return–Frames, d. h. dieser Zugriff erfolgt ebenfalls indirekt. Außerdem wird bei Peyton Jones wesentlich mehr Speicherplatz aufgrund der vergrößerten Sprungtabellen benötigt.

Ein Vergleich des Return–Mechanismus' zeigt einen kleinen Vorteil beim Zugriff auf eine von mehreren Alternativen bei Peyton Jones. Dort erfolgt der Zugriff auf die Adresse der richtigen Alternative über den Tabellenzeiger, während hier zunächst die Standard–Return–Funktion angesprungen werden muß. Dies wird jedoch ausgeglichen, wenn nur eine Alternative vorhanden ist[6]. Dann muß Peyton Jones über den Tabellenzeiger auf die Alternative zugreifen, während hier die Adresse bereits im Return–Frame enthalten ist.

5.6 Der Heap

Der Heap dient zur Aufnahme dynamisch erzeugter Strukturen (Ausdrücke und Daten).

Er besteht aus aufeinanderfolgenden 32-Bit Speicherzellen (im folgenden **Heapzellen** genannt), die ähnlich wie ein Stack verwaltet werden. Ein Zeiger (`HP`) weist

[6] `case`–Anweisungen mit nur einer Alternativen kommen nicht selten vor. Außer bei algebraischen Datentypen, die nur einen Konstruktor besitzen, existiert auch bei jeder `case`–Anweisung, die einen primitiven Datentyp auswertet nur eine Alternative (die dann mit Hilfe von *if–else*–Ketten das Pattern Matching vornimmt)

dabei auf die erste freie Zelle im Heap. Neue Heapknoten werden immer an dieser Stelle in den Heap eingefügt, wobei `HP` entsprechend der Größe des Heapknotens korrigiert wird. Erreicht der Zeiger `HP` das Ende des Heaps, das durch den Zeiger `HLimit` markiert wird, so wird eine Garbage–Collection durchgeführt. Der Heap wächst im Speicher nach unten. Dies ist notwendig, um für dynamisch erzeugte Heapknoten die gleiche Struktur zu erhalten, wie bei statisch erzeugten Heapknoten.

5.6.1 Statisch und dynamisch erzeugte Heapknoten

Funktionen des STG–Programms, die mit Hilfe des `let(rec)`–Befehls lokal innerhalb einer Funktion f definiert werden, werden erst bei der Reduktion der Funktion f, also während des Programmlaufs, dynamisch im Heap erzeugt. Um Funktionen, die auf oberster Ebene, also global, definiert wurden, nicht zu Beginn des erzeugten C–Programms zur Laufzeit im Heap erzeugen zu müssen, werden sie durch statische C–Strukturen dargestellt. Diese C–Strukturen werden implizit initialisiert, wodurch keine Auswirkungen auf die Laufzeit entstehen (vgl. Kernighan, Ritchie [KR83]). Diese *statischen Heapknoten* existieren also nicht im Heap. Während der Programmausführung werden Zeiger auf Heapknoten verwendet. Da nicht immer feststellbar ist, ob es sich dabei um einen Zeiger auf einen statischen oder einen dynamischen Heapknoten handelt, müssen beide Knotentypen die gleiche Struktur besitzen. Da der Zugriff auf die Komponenten einer C–Struktur durch Addition eines *positiven* Offsets zu deren Zeiger erfolgt, muß auch der Zugriff auf die Komponenten der dynamischen Heapknoten durch Addition eines positiven Offsets erfolgen. Hieraus folgt, daß der Heap im Speicher nach unten wachsen muß[7].

5.6.2 Die Struktur der Heapknoten

5.6.2.1 Die in den Heapknoten enthaltenen Informationen

Heapknoten repräsentieren STG–Funktionen bzw. Datenobjekte. Der C–Code zur Ausführung dieser Funktionen wird bei der Übersetzung des STG–Programms erzeugt. Die Heapknoten enthalten einen Zeiger auf diesen Code. Um nun einen Ausdruck, dargestellt durch einen Heapknoten zu reduzieren, muß die C–Funktion, deren Adresse (Zeiger) im Heapknoten enthalten ist, ausgeführt werden. Diese C–Funktionen werden im weiteren **Reduktionsfunktionen** genannt.

[7]Eigenlich folgt daraus nur, daß ein Heapknoten im Heap nach oben wachsen muß, der Zeiger auf diesen Heapknoten also auf dessen „unteres“ Ende zeigen muß. Dies bedeutet aber für einen nach oben wachsenden Heap, daß sich der Beginn eines neu erzeugten Heapknotens (dessen Adresse benötigt wird) mehrere Zellen unterhalb von `HP` befindet (entspechend der Größe des Heapknotens). Bei nach unten wachsendem Heap befindet sich der Beginn jedoch direkt oberhalb von `HP`. Der Einfachheit halber wurde deshalb ein nach unten wachsender Heap gewählt.

Auch Heapknoten, die Datenobjekte (Konstruktoren) darstellen, enthalten eine solche Reduktionsfunktion. Diese Funktion lädt lediglich die Return–Register mit dem Datenobjekt (vgl. Regel 7). In der Tat sind in den Heapknoten zwei Zeiger auf Reduktionsfunktionen enthalten. Die erste Reduktionsfunktion überprüft zunächst, ob genügend Argumente zum Ausführen der Funktion vorhanden sind (**Standardeinsprung**), während die zweite Reduktionsfunktion davon ausgeht, daß alle benötigten Argumente vorhanden sind (**direkter Einsprung**). Freie Variablen, die zur Reduktion eines Ausdrucks nötig sind[8], sind bereits bei der Erzeugung des Heapknotens, der diesen Ausdruck darstellt, vorhanden. Die Reduktion dieses Ausdrucks kann aber zu einem späteren Zeitpunkt erfolgen, so daß die freien Variablen zwischengespeichert werden müssen. Diese Zwischenspeicherung erfolgt im Heapknoten.

Auch die Argumente einer partiellen Applikation und Datenobjekte (Konstruktoren mit Komponenten) werden in den Heapknoten gespeichert. Zusätzlich werden in jedem Heapknoten Informationen benötigt, die es der Garbage–Collection erlaubt, jeden Knoten korrekt zu behandeln (z. B. Knotenlänge etc.).

Bei bereits existierenden Übersetzern für funktionale Sprachen hat sich eine Knotenstruktur durchgesetzt, die aus einem dynamischen Teil, der im Heap erzeugt wird, und einem statischen Teil, der während der Übersetzung erzeugt wird, besteht. Der dynamische Teil (im weiteren ebenfalls **Heapknoten** genannt) enthält einen Zeiger auf den statischen Teil (im weiteren **Info–Tabelle** genannt). Die während der Übersetzung erzeugte Info–Tabelle reduziert den Aufwand zur Erzeugung eines Heapknotens zur Laufzeit auf das Anlegen des dynamischen Teils. Allerdings erfolgt der Zugriff auf die Informationen in der Info–Tabelle indirekt, über dessen Zeiger im Heapknoten. Aus diesem Grund ist nicht sofort klar, welche Informationen im statischen und welche Informationen im dynamischen Teil des Heapknotens gehalten werden[9]. Auch der Aufbau des dynamischen Heapknotenteils kann unterschiedlich sein, wie ein Vergleich der Heapknoten bei Peyton Jones [Pey92] und Clean [SNGP91] zeigt (s. Abb. 5.3).

[8] Die STG–Sprache besitzt im Gegensatz zu anderen funktionalen Sprachen die Fähigkeit, freie Variablen ohne vorausgehendes *lambda lifting* zu manipulieren.

[9] Abgesehen von Informationen (wie z. B. den freien Variablen), die erst zur Laufzeit zur Verfügung stehen und deshalb im dynamischen Heapknotenteil gespeichert werden

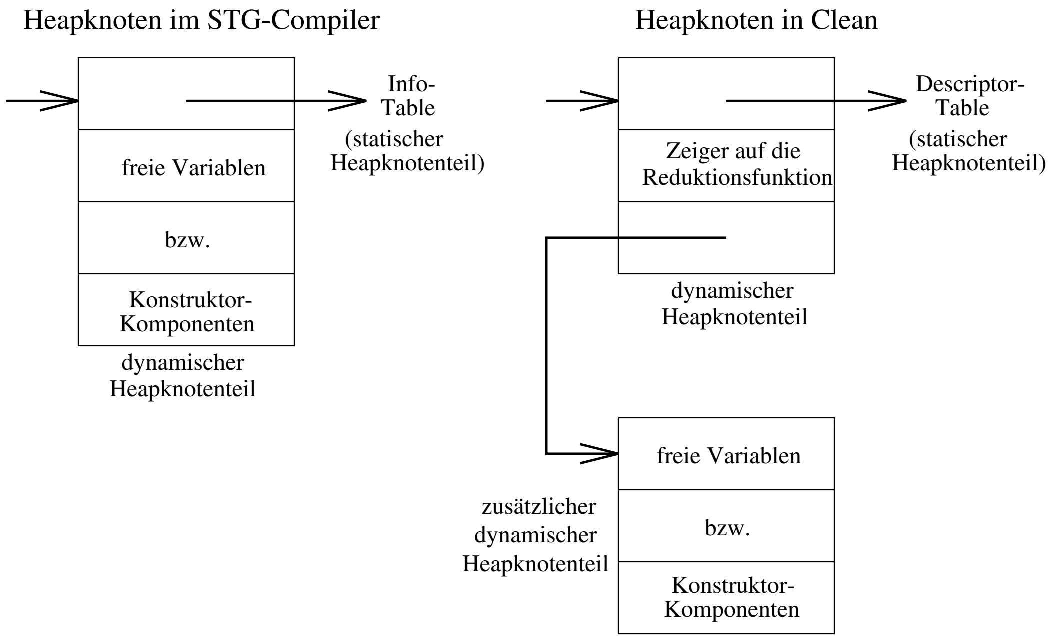


Abbildung 5.3: *Der Aufbau der Heapknoten im STG–Compiler und in Clean*

So wird z. B. der Zeiger auf die Reduktionsfunktion bei Clean im dynamischen Teil des Heapknotens gehalten, während dieser Zeiger bei Peyton Jones im statischen Teil gespeichert wird. Dadurch wird ein schnellerer Zugriff auf die Reduktionsfunktion erreicht, was jedoch mit zusätzlichem Aufwand bei der Erzeugung des Heapknotens verbunden ist.

Das Speichern der freien Variablen in einem gesonderten (dynamischen) Heapknoten, wie bei Clean, zielt hingegen auf eine Vereinfachung der Update–Funktion ab. So muß ein Heapknoten, der nach seiner Reduktion mit einem Ergebnis überschrieben werden soll, das größer als der Heapknoten selbst ist, nicht komplett neu erzeugt werden. Vielmehr kann der eigentliche Heapknoten (mit dem Zeiger auf die Info–Tabelle) weiterverwendet werden. Es muß lediglich der variable Heapknotenteil (mit den freien Variablen) neu erzeugt und der Zeiger auf diesen Teil im eigentlichen Heapknoten verändert werden. Es entsteht also kein **Indirection Node**[10]. Dies ist jedoch mit einem indirekten Zugriff auf die freien Variablen verbunden.

Zur Bestimmung der „optimalen" Heapknotenstruktur werden deshalb einige Testprogramme (s. Anhang) mit den verschiedenen Strukturen verglichen.

[10] Bei Peyton Jones muß hingegen ein komplett neuer Knoten angelegt werden. Da die Adresse des alten Knotens jedoch noch an anderen Stellen im Programm verwendet werden kann, muß der alte Knoten mit einem Verweis auf den neu angelegten Knoten überschrieben werden. Diesen Verweis bezeichnet man als *Indirection Node*

5.6.3 Optimierung der Heapknotenstruktur bzgl. des Zugriffs auf die Reduktionsfunktion

Die Speicherung der Zeiger auf die Reduktionsfunktionen im dynamischen Teil des Heapknotens verspricht einen schnelleren Zugriff auf diese Funktion und somit eine geringere Laufzeit des gesamten Programms. Allerdings ist dies mit einem Mehraufwand bei der Erzeugung der Heapknoten und bei der Garbage–Collection verbunden (größere Heapknoten). Um Vor– und Nachteile dieser Heapknotenstruktur mit einer Struktur zu vergleichen, bei der die Zeiger auf die Reduktionsfunktionen in der Info–Tabelle gespeichert sind, werden einige Programme mit den beiden Heapknotenstrukturen A und B in Abb. 5.4 verglichen.

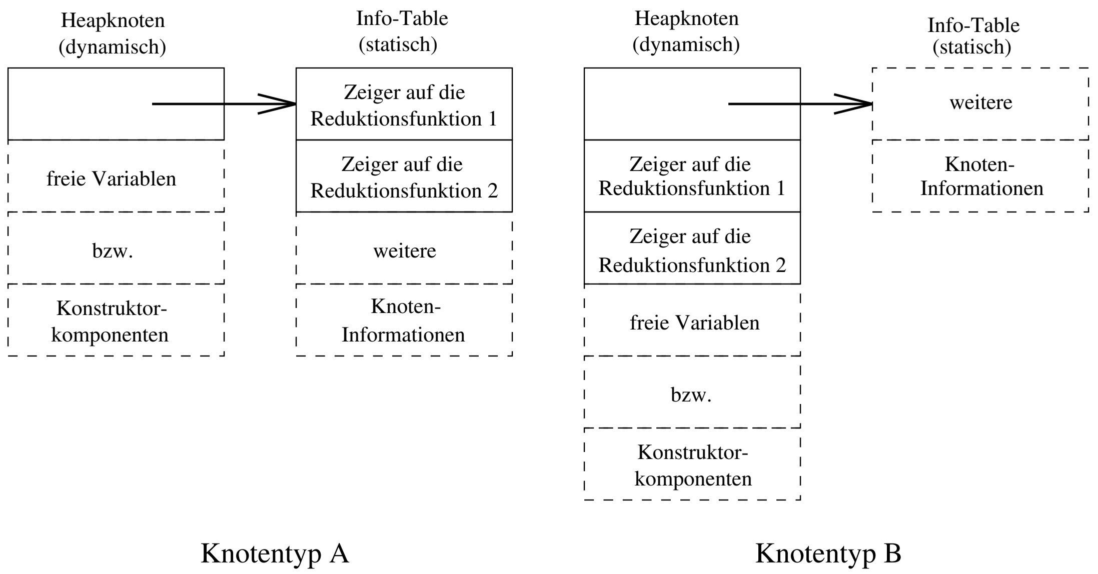


Abbildung 5.4: *Heapknotenaufbau mit Code–Pointern im dynamischen und im statischen Heapknotenteil: Reduktionsfunktion 1 bezeichnet die Funktion, die zunächst überprüft, ob genügend Argumente vorhanden sind (Standardeinsprung); Reduktionsfunktion 2 geht davon aus, daß alle zur Reduktion benötigten Argumente vorhanden sind (direkter Einsprung).*

Zunächst wird die Takeuchi–Funktion betrachtet (s. Anhang B). Diese Funktion verwendet ausschließlich primitive Werte (unboxed values) und benötigt keinen Heapspeicher. Hierdurch kommen lediglich die Vorteile des Knotentyps B, nämlich der schnellere Zugriff auf die Reduktionsfunktionen, zum Tragen. Die Nachteile des Knotentyps B, bei der Erzeugung von Heapknoten und deren Garbage–Collection (bedingt durch den vergrößerten dynamischen Knotenteil), entstehen hier nicht. Die Takeuchi–Funktion stellt also den *best case* für den Knotentyp B dar.

Takeuchi	benötigt Heapzellen	Heapgröße	Laufzeit
Heapknotentyp A	0	100000	16.95s
Heapknotentyp B	0	100000	16.26s

Tabelle 5.3: *Laufzeitvergleich der Takeuchi–Funktion bei der Verwendung der Heapknotenstrukturen A und B*

Wie Tabelle 5.3 zeigt, ergibt sich bei der Speicherung der Reduktionsfunktion im dynamischen Teil des Heapknotens (Typ B) im *best case* ein Zeitgewinn von nur ca. 4%. Als nächstes wird das Programm *Primes*, zur Berechnung der zehntausendsten Primzahl mit Hilfe des Siebs des Eratosthenes betrachtet. Dieses Programm hat einen sehr großen Bedarf an Heapspeicher, da alle berechneten Primzahlen in Form von algebraischen Daten (boxed values) im Heap gehalten werden müssen.

Primes	Heapknotentyp A	Heapknotentyp B
Heapgröße	100010	100010
benötigte Heapzellen	60004	100009
Garbage–Collections	0	0
Laufzeit (s)	27.16	26.97

Tabelle 5.4: *Laufzeitvergleich des Programms „Primes" bei der Verwendung der Heapknotenstrukturen A und B*

Wie Tabelle 5.4 zeigt, geht beim Knotentyp B der Laufzeitgewinn bei der Verwendung von Heapspeicher wieder verloren (Laufzeitunterschied unter 1%). Dies geht auf die vergrößerten Heapknoten zurück, bei deren Erzeugung jeweils zwei Speicherzellen zusätzlich beschrieben werden müssen. Besonders deutlich wird der Unterschied zwischen den Heapknotentypen A und B bei einem Programm zur Berechnung des einmillionsten Elementes der Liste der natürlichen Zahlen. Wie Tabelle 5.5 zeigt benötigt dieses Programm sehr viel Heapspeicher, wodurch auch Garbage–Collections nötig werden.

	Heapknotenstruktur A	Heapknotenstruktur B
Heapgröße	100000	100000
benötigte Heapzellen	7999998	13999999
Garbage–Collections	80	140
Laufzeit	34.69s	38.20s

Tabelle 5.5: *Laufzeitvergleich eines Programms zur Berechnung des einmillionsten Elementes der Liste der natürlichen Zahlen*

Da es sich bei Programmen wie der Takeuchi–Funktion, die keinen Heapspeicher benötigen, eher um Ausnahmen handelt, werden im folgenden die Zeiger auf die Reduktionsfunktionen nicht im dynamischen Teil des Heapknoten sondern in der Info–Tabelle gespeichert (Heapknotentyp A).

5.6.4 Optimierung der Heapknotenstruktur bzgl. der Update–Funktion

Das Speichern der freien Variablen in einem gesonderten Heapknoten, wie bei Clean (s. Abb. 5.3), vereinfacht das Überschreiben dieses Knotens mit einem Ergebnis, das größer ist als der Knoten selbst. In diesem Fall muß bei der Verwendung des Knotentyps A ein komplett neuer Knoten erzeugt werden und der alte Knoten durch einen Indirection Node ersetzt werden. Bei Clean muß nur der gesonderte Teil mit den freien Variablen neu erzeugt werden. Der eigentliche Heapknoten, dessen Adresse im Programm benutzt wird, kann weiter verwendet werden. Hierbei wird also die Initialisierung des Indirection Nodes gespart, wodurch bei der weiteren Verwendung des Knotens der Zugriff auch nicht indirekt über diesen Indirection Node erfolgt. Außerdem muß die Garbage–Collection diese Indirection–Nodes nicht gesondert behandeln (d. h. entfernen). Alle diese Vorteile werden jedoch durch den indirekten Zugriff auf die freien Variablen eines Heapknotens und die (um den Zeiger auf den Variablenteil) vergrößerten Heapknoten erkauft.

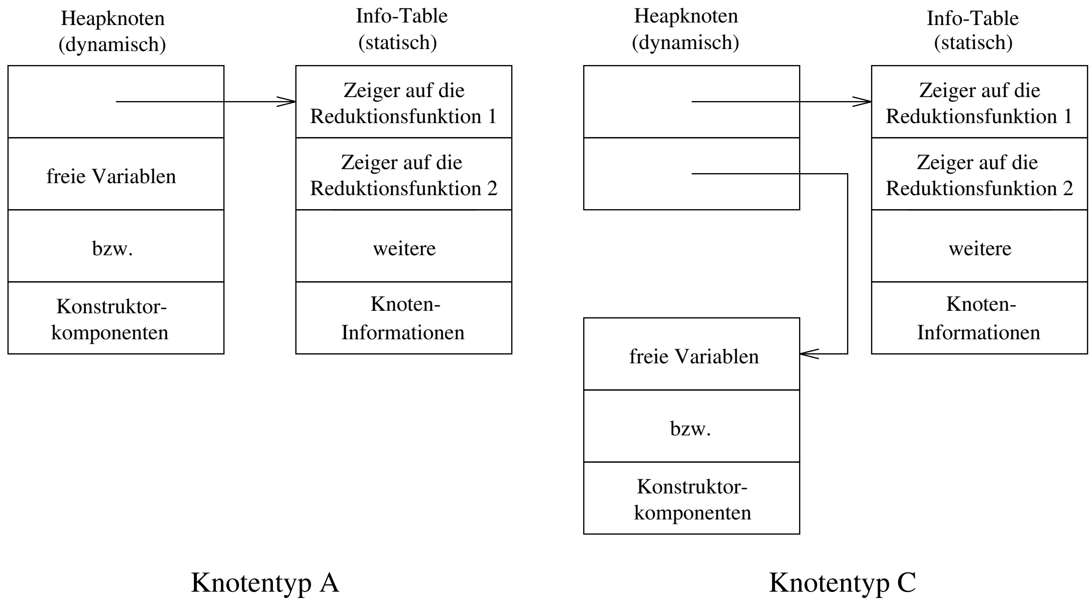


Abbildung 5.5: *Heapknotenaufbau mit freien Variablen im dynamischen und im statischen Heapknotenteil*

Um Auswirkungen dieser Art der Variablenspeicherung (im folgenden Heapknotenstruktur C genannt, s. Abb. 5.5) auf die Laufzeit und die benötigte Heapgröße zu erkennen, wird wieder das Programm zur Berechnung des einmillionsten Elementes der Liste der natürlichen Zahlen betrachtet, da bei diesem Programm nur Updates auftreten, bei denen das Ergebnis größer ist als der Knoten, der überschrieben werden soll (also beim Heapknotentyp A Indirection Nodes erzeugt werden müssen). D. h. dieses Programm stellt also den *best case* für den Heapknotentyp C im Vergleich zum Typ A dar.

	Heapknotentyp A	Heapknotentyp C
Heapgröße	100000	100000
benötigte Heapzellen	7999998	9000000
Garbage–Collections	80	90
Updates 'in place'	0	0
Updates 'in new cells'	999999	999999
Laufzeit	34.69s	34.50s

Tabelle 5.6: *Laufzeitvergleich bei der Verwendung der Heapknotentypen A und C*

Wie Tabelle 5.6 zeigt, benötigen die beiden Versionen in etwa die gleiche Ausführungszeit. D. h. der Mehraufwand, der bei der Erzeugung und Verfolgung der Indirection Nodes beim Knotentyp A entsteht, wird durch den indirekten Zugriff auf die freien Variablen beim Knotentyp B ausgeglichen. Im schlechtesten Fall, also wenn alle Heapknoten groß genug sind, um bei einem Update ihr Ergebnis aufzunehmen (*updates in place*) und somit keine Indirection Nodes entstehen, ergibt sich ein Nachteil des Heapknotentyps C gegenüber dem Typ A. Als Beispiel wird hier ein Programm zur Berechnung der zwanzigtausendsten Primzahl betrachtet, bei dem fast ausschließlich *updates in place* auftreten, wie Tabelle 5.7 zeigt.

Primes	Heapknotentyp A	Heapknotentyp C
Heapgröße	200000	200000
benötigte Heapzellen	120005	160004
Garbage–Collections	0	0
Updates 'in place'	19998	19998
Updates 'in new cells'	1	1
Laufzeit (abs.)	60.89s	66.07s
Laufzeit (rel.)	100%	108.5%

Tabelle 5.7: *Laufzeitvergleich bei der Verwendung der Heapknotentypen A und C*

Da bei der Verwendung des Heapknotentyps C auch solche Heapknoten, die nicht mit ihrem Ergebnis überschrieben werden sollen (nonupdatable), mit einem gesonderten Variablenteil ausgestattet sind, verspricht ein gemeinsame Verwendung der Knotentypen A und C eine bessere Laufzeit und einen geringeren Heapspeicherbedarf. Dabei sollten Heapknoten, die als *nonupdatable* gekennzeichnet sind, durch den Knotentyp A dargestellt werden und Heapknoten, die überschrieben werden sollen durch den Knotentyp C. Hierdurch verringert sich der benötigte Heapspeicher, da der Knotentyp A keinen Zeiger auf den Variablenteil benötigt. Die Laufzeit verringert sich auf Grund des direkten Zugriffs auf die Variablen und der geringeren Knotengröße. Allerdings erfolgt der Zugriff auf die freien Variablen eines zu überschreibenden Heapknotens (Typ C) immer noch indirekt.

Vorteile, die durch die gemeinsame Verwendung der Knotentypen A und C gegenüber der ausschließlichen Verwendung des Knotentyps A entstehen, resultieren jedoch lediglich aus der Tatsache, daß keine Indirection Nodes entstehen. Ein ganz neuer Ansatz, der die Entstehung von Indirection Nodes bei der ausschließlichen Verwendung des Heapknotentyps A vermeidet, verspricht daher ein besseres Ergebnis.

Dies wird erreicht, indem die Heapknoten bei ihrer Erzeugung bereits so groß angelegt werden, daß sie bei einem Update immer mit ihrem Ergebnis überschrieben werden können (*updates in place*). Dieses Verfahren wird im folgenden als Verwendung des *Heapknotentyps A mit erweiterter Knotengröße* bezeichnet. Der

Platz, der für einen solchen Heapknoten benötigt wird, kann aus dem Typ der Funktion, die durch den Knoten dargestellt wird, bestimmt werden (hier gehen Typinformationen in die Code–Optimierung ein). Somit entstehen keine Indirection Nodes und alle freien Variablen können (im Gegensatz zum Knotentyp C) direkt angesprochen werden.

Allerdings wird hierbei Platz reserviert, obwohl nicht feststeht, ob dieser Platz überhaupt benötigt wird, d. h. ob der Update überhaupt ausgeführt wird. Außerdem muß stets Platz für das größtmögliche Ergebnis bereitgestellt werden. So kann z. B. der Typ einer Funktion ein algebraischer Datentyp sein, der aus zwei Konstruktoren besteht, wobei ein Konstruktor die Stelligkeit 0 besitzt, während der zweite Konstruktor die Stelligkeit 10 besitzt. Da das Ergebnis der Reduktion zum Zeitpunkt der Erzeugung des Heapknotens nicht bekannt ist, muß Heapspeicher für den 10-stelligen Konstruktor reserviert werden. Dieser Mehrbedarf an Heapspeicher kann zu einer vergrößerten Anzahl von Garbage–Collections führen, die die Vorteile dieses Verfahrens beeinträchtigen können. Deshalb folgende

5.6.4.1 Abschätzung des Heapspeicherbedarfs.

Folgende Abschätzung bstimmt den Heapspeicherbedarf bei einem Update bei den Heapknotentypen A, C und A mit erweiterter Knotengröße.

Sei n die Länge des zu überschreibenden Heapknotens (d. h. ein Zeiger auf die Info–Tabelle und $(n-1)$ freie Variablen) und m die Länge des Ergebnisknotens (ebenfalls ein Zeiger auf die Info–Tabelle und (m - 1) freie Variablen), mit $a \leq m \leq b$, wobei a die minimale und b die maximale Größe des Ergebnisknotens sind. Es gelte $a \leq n$, d. h. der zu überschreibende Knoten ist zu klein, um den neuen Knoten aufzunehmen.

- Bei der Verwendung des Heapknotentyps A wird bei einem Update der Ergebnisknoten komplett neu erzeugt. Der Heapspeicherbedarf beträgt also $n+m$ (alter und neuer Knoten). Außerdem muß an der Stelle des zu überschreibenden Heapknotens ein Indirection Node erzeugt werden.

- Bei der Verwendung des Heapknotentyps C besitzt der zu überschreibende Heapknoten die Größe $n+1$ aufgrund des zusätzlich benötigten Zeigers auf den Variablenteil. Bei einem Update muß jedoch nur der Variablenteil der Länge $m-1$ neu erzeugt werden, woraus sich ebenfalls ein Heapspeicherbedarf von $n+m$ ergibt. Hierbei entstehen keine Indirection Nodes.

- Bei der Verwendung des Knotentyps A mit erweiterter Knotengröße, ist die Größe des zu überschreibenden Heapknotens b, um ihn durch jedes mögliche Ergebnis ersetzen zu können.

Im schlechtesten Fall, also wenn der Knoten mit einem Ergebnis minimaler Länge überschrieben werden soll ($m = a$), benötigt die letzte Version nur dann mehr Heapspeicher, falls $b > n + a$ gilt.

Da sowohl die Heapknotengröße n, als auch die minimale und maximale Ergebnisgröße a und b schon zur Übersetzungszeit bekannt sind, kann bereits hier entschieden werden, ob ein zu erzeugender Knoten so groß gewählt werden soll, damit er mit seinem Ergebnis überschrieben werden kann, oder ob bei einem Update neuer Platz beschafft werden soll.

Somit werden die als nonupdatable markierten Knoten durch den Knotentyp A dargestellt und zu überschreibende Knoten für die $b > n + a$ gilt, ebenfalls durch diesen Typ, wobei jedoch der Knoten (entsprechend seinem Ergebnis) vergrößert wird. Die restlichen zu überschreibenden Knoten (dies dürfte ein relativ kleiner Prozentsatz sein) werden im folgenden durch den Knotentyp A dargestellt.

Zum Vergleich wieder das Programm zur Bestimmung des einmillionsten Elementes der Liste der natürlichen Zahlen.

	Knotentyp A	Knotentyp C	A mit erweiterter Knotengröße
Heapgröße	100000	100000	100000
benötigte Heapzellen	7999999	9000000	6000003
Garbage–Collections	80	90	60
Updates 'in place'	0	0	999999
Updates 'in new cells'	999999	999999	0
Laufzeit (abs.)	34.69s	34.50s	32.19s
Laufzeit (rel.)	100%	99.5%	92.8%

Tabelle 5.8: *Laufzeitvergleich bei der Berechnung des einmillionsten Elements der Liste der natürlichen Zahlen*

Wie Tabelle 5.8 zeigt, ergibt die Verwendung des Knotentyps A mit erweiterter Knotengröße deutlich effizientere Programme. Wird für Knoten, für die eine Erweiterung der Größe auf Grund des Mehrverbrauchs an Heapspeicher nicht in Frage kommt, der Knotentyp A gewählt, sind die Laufzeit und der Heapspeicherbedarf auch im schlechtesten Fall nicht größer als bei der ausschließlichen Verwendung des Heapknotentyps A. Deshalb wird im weiteren der Heapknotentyp A mit erweiterter Knotengröße verwendet.

5.7 Die globale Umgebung σ

Um global definierte STG–Funktionen im gesamten STG–Programm verwenden zu können, bildet die globale Umgebung σ die Namen der Stg–Funktionen auf deren Adressen im Heap ab (bzw. auf C–Strukturen, vgl. Abschnitt 5.6.1)[11]. Diese

[11] Im Compiler werden tatsächlich noch einige andere, zur Übersetzung benötigte Informationen, wie z. B. die Stelligkeit der globelen Funktionen, in der globalen Umgebung gehalten.

Informationen werden jedoch nur bei der Übersetzung des STG–Programms in C–Code benötigt. Das resultierende C–Programm enthält also keine Komponente, die der globalen Umgebung der operationalen Semantik entspricht.

5.8 Der Code

Die vier Befehle *Eval*, *Enter*, *ReturnCon* und *ReturnInt* der operationalen Semantik werden nicht direkt (etwa als Funktionen) in das zu erzeugende C–Programm eingefügt. Vielmehr wird deren Funktionalität in der entsprechenden C–Funktion durch C–Code dargestellt.

So kann z. B. der Befehl *Enter* a die Funktion, die durch den Heapknoten an der Adresse a dargestellt wird, auf deren Argumente anwenden, indem der im Heapknoten enthaltene Zeiger auf die Funktion verwendet wird. Handelt es sich jedoch um eine bekannte Funktion, so kann diese u. U. direkt angesprungen werden. Dies zeigt, daß eine Abbildung dieser Befehle auf feststehenden C–Code ineffizient ist. (Zur genauen Abbildung von STG–Programmen auf C–Programme s. Kapitel 6).

5.9 Die lokale Umgebung ρ

Die lokale Umgebung ρ dient bei einer Funktionsanwendung zur Aufnahme der Funktionsargumente und der freien Variablen (vgl. z. B. Regel 3) und nach einer `case`–Anweisung zur Aufnahme der Konstruktorkomponenten (vgl. Regel 7).

In der Implementierung existiert diese lokale Umgebung jedoch nicht als eigenständige Komponente. Somit bleiben die Argumente aus den Stacks liegen, die freien Variablen im Heapknoten und die Konstruktorkomponenten in den Registern und werden bei Bedarf von dort gelesen. Dies erspart das Kopieren dieser Werte. Allerdings bringt diese *verteilte* lokale Umgebung auch einige Probleme mit sich. So muß z. B. am „Ende“ einer Funktion dafür gesorgt werden, daß deren Argumente von den Stacks entfernt werden. (In der operationalen Semantik erfolgt dies schon zu Beginn der Funktionsauswertung, indem die Argumente in die lokale Umgebung ρ eingefügt werden).

Kapitel 6

Die Abbildung von STG–Funktionen auf C

Jede STG–Funktion wird im zu erzeugenden C–Programm durch folgende Komponenten dargestellt:

- einen Heapknoten
- eine Informationstabelle (Info–Tabelle)
- und eine oder zwei Reduktionsfunktionen

Der Heapknoten enthält einen Zeiger auf die Informationstabelle, die wiederum zwei Zeiger auf die beiden Reduktionsfunktionen enthält. (Zum genauen Aufbau der Heapknoten und der Informationstabellen s. Abschnitt 5.6.2). Die Manipulation einer STG–Funktion erfolgt im erzeugten C–Programm über den Zeiger auf den zugehörigen Heapknoten. Über diesen Zeiger können die Reduktionsfunktionen aufgerufen werden, er gestattet den Zugriff auf die freien Variablen (die sich im Heapknoten befinden), er läßt sich in Datenstrukturen (als Konstruktorkomponente) einfügen usw..

6.1 Die Erzeugung der Heapknoten

Bei einer globalen STG–Funktion wird der Heapknoten durch eine C–Struktur dargestellt, die bereits beim Übersetzen erzeugt wird. Die Initialisierung dieser Struktur erfolgt also durch den Linker des C–Compilers und hat keine Auswirkungen auf die Laufzeit (vgl. Abschnitt 5.6.1).

Die Heapknoten lokal definierter STG–Funktionen hingegen werden erst zur Laufzeit (genauer: während der Reduktion des `let(rec)`–Ausdrucks in dem die STG–Funktionen definiert sind) im Heap erzeugt.

6.2 Die Erzeugung der Informationstabellen

Die Informationstabellen, sowohl global als auch lokal definierter STG–Funktionen, werden auch durch C–Strukturen dargestellt, d. h. während des Übersetzens erzeugt. Die Initialisierung der Informationstabellen hat also ebenfalls keinen Einfluß auf die Laufzeit des resultierenden C–Programms.

6.3 Die Reduktionsfunktionen

Die Reduktion einer Funktion besteht aus mehreren Teilen:

- Zunächst muß überprüft werden, ob genügend Argumente zur Reduktion auf den Argument–Stacks vorhanden sind (*Argumentsättigungstest*). Diese Überprüfung kann entfallen, falls die Funktion keine Argumente besitzt oder übersprungen werden, wenn der Übersetzer feststellen kann, daß die Funktion auf eine ausreichende Anzahl von Argumenten angewandt wird.
- Eine als updatable gekennzeichnete Funktion (also eine Funktion, deren Heapknoten später durch ihren Wert ersetzt werden soll), muß einen Update–Frame auf dem B–Stack erzeugen.
- Eine Funktion, die die Stacks verwendet muß die Stacks auf einen Überlauf testen (*Stack overflow check*).
- Der Zeiger auf die Informationstabelle einer als updatable gekennzeichnete Funktion kann vor ihrer Reduktion zur Fehlererkennung verändert werden.
- Und schließlich muß die eigentliche Funktion, d. h. der in der Funktion definierte Ausdruck ausgewertet werden.

Alle diese Teile werden auf zwei C–Funktionen aufgeteilt. Der Argumentsättigungstest bildet eine C–Funktion, den *Standardeinsprung*, alle anderen Teile bilden zusammen die zweite C–Funktion, den *direkten Einsprung*. Die Aufteilung in zwei C–Funktionen erfolgt, da der Argumentsättigungtest evtl. übersprungen werden kann (s. o.). Da es in C jedoch nicht möglich ist, zu einer beliebigen Anweisung innerhalb einer Funktion zu springen, wird dieser Test in einer eigenen Funktion dargestellt.

Standardeinsprung und direkter Einsprung stellen die beiden Reduktionsfunktionen dar, auf die im Heapknoten bzw. in der Informationstabelle verwiesen wird.

6.3.1 Der Typ der Reduktionsfunktionen

Die Argumentübergabe an Funktionen erfolgt mit Hilfe der Argument–Stacks. Auf die Verwendung des C–Parameterübergabemechanismuses wird verzichtet (vgl. Abschnitt 5.1.2). Die Reduktionsfunktionen erwarten also keine (C–)Argumente.

Der Rückgabewert der Reduktionsfunktionen ist stark von der Laufzeitumgebung abhängig. Um eine einfache Anpassung an die jeweilige Laufzeitumgebung zu ermöglichen, endet jede Reduktionsfunktion mit einem der folgenden drei Makros:

`JUMP (f);` Die Funktion `f` soll angesprungen werden.

`ENTER (a);` Die Funktion, die durch den Heapknoten an der Adresse `a` repräsentiert wird, soll aufgerufen werden.

`RETURN;` Es soll mit der Funktion fortgefahren werden, deren Adresse sich im obersten Return–Frame bzw. Update–Frame auf dem B–Stack befindet.

Diese Funktionen können nun direkt angesprungen werden (z. B. mit in das C–Programm eingefügten Assemblerbefehlen). In diesem Fall liefern die Reduktionsfunktionen keinen Wert zurück. Eine andere Möglichkeit besteht darin, die Adresse der aufzurufenden Funktion an einen *Ein–Zeilen–Interpreter* zurückzugeben, der diese Funktion dann aufruft. In diesem Fall liefern die Reduktionsfunktionen einen Zeiger auf eine Funktion zurück (s. Abschnitt 7.4.8).

6.3.2 Der Standardeinsprung

Der Standardeinsprung überprüft zunächst, ob genügend Argumente zur Reduktion der Funktion auf den Argument–Stacks vorhanden sind (Argumentsättigungstest). Obwohl die Argumente auf den A– und den B–Stack aufgeteilt sind, wird lediglich die Anzahl der Argumente auf einem, nämlich dem Stack überprüft, auf dem das letzte Argument der Funktion erwartet wird. Ist das letzte Argument vorhanden, so sind es sicher auch alle anderen (zur Bestimmung der Anzahl der auf einem Stack vorhandenen Argumente s. Abschnitt 5.1.1). Sind alle Argumente vorhanden, so wird zum direkten Einsprung verzweigt. Ansonsten stellt der gerade reduzierte Ausdruck (also die Anwendung der Funktion auf eine nicht ausreichende Anzahl von Argumenten) eine partielle Applikation dar. Dann erfolgt ein Update, der eine zuvor angewandte, als updatable markierte Funktion (bzw. deren Heapknoten) mit dieser partiellen Applikation überschreibt (vgl. Regel 4).

Der Name dieser Reduktionsfunktion besteht aus dem Namen der entsprechenden STG–Funktion und dem Suffix `_entry`.

Der Standardeinsprung der folgenden STG–Funktion

$$\texttt{f = } \{\} \ \backslash\pi \ \{\texttt{a\#,b\#,c}\} \texttt{ -> } e$$

lautet also:

```
unsigned *f_entry(void)
{
        if ( (SBaseA - SpA) < 1 )
                {JUMP ( UpdatePAPA );}
        else
                {JUMP ( f_direct );}
}
```

(`UpdatePAPA` ist dabei der Name der Funktion, die die Aktualisierung im Fall einer partiellen Applikation (s. o.) vornimmt und `f_direct` ist der Name des direkten Einsprungs (s. u.)) Benötigt die zu übersetzende STG–Funktion keine Argumente zu ihrer Reduktion, so kann auf die Erzeugung des Standardeinsprungs verzichtet werden. In diesem Fall verweisen beide Funktionszeiger in der Informationstabelle auf den direkten Einsprung.

6.3.3 Der direkte Einsprung

Wie in Abschnitt 6.3 beschrieben, besteht der direkte Einsprung aus mehreren Teilen. Diese Teile werden im folgenden beschrieben.

6.3.3.1 Die Erzeugung eines Update–Frames

Funktionen, die als updatable gekennzeichnet sind (also die Markierung `\u`), erzeugen bei ihrer Reduktion einen Update–Frame auf dem B–Stack (vgl. Regel 2). Dieser Frame besteht aus der Adresse des zu überschreibenden Heapknotens, den beiden Basiszeigern der Argument–Stacks und einem Zeiger auf die Standard–Update–Funktion (vgl. Abschnitt 5.4.1). Die Erzeugung dieses Rahmens sieht in C wie folgt aus (dabei zeigt `Node` auf den gerade zu reduzierenden Heapknoten, und `Update_REG` ist der Name der Standard–Update–Funktion):

```
SpB[0] = Node;      /* Update-Frame */
SpB[1] = SBaseB;
SpB[2] = SBaseA;
SpB[3] = Update_REG;
SpB += 4;
```

6.3.3.2 Der Stack–Überlauftest

Da sich der A– und der B–Stack einander entgegenwachsen, erfolgt ein Stack–Überlauftest, indem die beiden Zeiger auf die erste freie Speicherzelle der Stacks (`SpA` und `SpB`) miteinander verglichen werden. Da der A–Stack im Speicher nach unten und der B–Stack nach oben wächst, tritt ein Stack–Überlauf ein, falls die Differenz aus `SpA` und `SpB` negativ wird.

Der Stack–Überlauftest wird jedoch nicht immer dann ausgeführt, wenn gerade ein Element auf einen Stack gelegt wird, sondern nur einmal zu Beginn der Funktionen, die die Stacks verwenden. Hierzu muß die Anzahl der von der Funktion benötigten Stack–Speicherplätze berechnet werden. Bei dieser Berechnung können alle Alternativen einer `case`–Anweisung berücksichtigt werden. Die Alternative mit dem größten Stack–Speicherbedarf (*worst case*) wird dann zur Berechnung des gesamten Speicherbedarfs herangezogen.

Der Stack–Überlauftest für eine Funktion, die insgesamt fünf Elemente auf den Stack legen will (für den Test spielt es dabei keine Rolle, auf welchen Stack die Elemente gelegt werden), sieht also wie folgt aus:

```
if ( SBaseA - 5 < SBaseB)
    ERROR(STACK_OVERFLOW);
```

Die Funktion `ERROR` beschreibt dabei eine Funktion der Laufzeitumgebung, die eine entsprechende Fehlermeldung ausgibt und das Programm abbricht.

6.3.3.3 Fehlererkennung durch Änderung des Zeigers auf die Informationstabelle

Wird eine als updatable gekennzeichnete Funktion reduziert, so besteht die Möglichkeit, den Zeiger im Heapknoten, der auf die zugehörige Informationstabelle verweist, für die Dauer der Reduktion zu verändern. Indem dieser Zeiger auf eine spezielle Informationstabelle zeigt, die anstelle der Reduktionsfunktionen Funktionen zur Ausgabe einer Fehlermeldung enthält, kann sobald diese Funktion ein erneutes Mal reduziert werden soll (dies wäre ein Fehler im Programm, da die Funktion dann von sich selbst abhängt), eine Fehlermeldung erzeugt werden. Wird dieser Zeiger nicht verändert, führt dieser Fehler zu einem Stack–Überlauf.

In einem parallelen System kann dieser Zeiger während der Reduktion so verändert werden, daß ein anderer Prozeß, der versucht diesen Heapknoten zu reduzieren, bis zu dessen Update unterbrochen wird.

6.4 Die Übersetzung der STG–Ausdrücke

Im folgenden wird nun die Übersetzung der einzelnen STG–Ausdrücke beschrieben.

6.4.1 Die Übersetzung einer Funktionsanwendung

Die Anwendung einer Funktion f besteht laut Regel 1 aus zwei Teilen:

- Zunächst müssen die Argumente, auf die die Funktion f angewandt werden soll, auf die Argument–Stacks gelegt werden.
- Dann kann der Heapknoten, der die Funktion f darstellt, ausgewertet werden.

6.4.1.1 Die Argumentübergabe

Die Übergabe von Argumenten an eine Funktion erfolgt auf den Argument–Stacks. Hierzu müssen die Argumente auf den A– und den B–Stack aufgeteilt werden (zur Erinnerung: der A–Stack enthält Zeiger auf Heapknoten; der B–Stack primitive Werte wie Integer, Characters, usw.). Diese Aufteilung der Argumente ist in der hier verwendeten STG–Syntax sehr einfach. Im Gegensatz zur STG–Syntax, die bei Peyton Jones [Pey92] verwendet wird, besitzen hier alle Namen von primitiven Werten (unboxed values), also auch Argumente, die auf dem B–Stack übergeben werden müssen, das Suffix `#` (vgl. Abschnitt 3.2.4).

Ein größeres Problem bei der Argumentübergabe stellt die lokale Umgebung ρ dar. Diese besteht (teilweise) aus Elementen der Argument–Stacks (vgl. Abschnitt 5.9) und geht bei einer Funktionsanwendung verloren (vgl. Regel 1). D. h. ein Teil der Elemente auf den Argument–Stacks muß vor der Argumentübergabe entfernt bzw. durch die Argumente überschrieben werden. Dabei ist jedoch darauf zu achten, daß ρ die Werte der zu übergebenden Argumente enthalten kann. So muß z. B. bei folgender Funktion

```
f = {} \n {a,b,c#,d#,e#} = g {b,a,d#,c#}
```

bei der, während der Ausführung, die Argumente `a`, `b`, `c#`, `d#` und `e#` auf den Stacks liegen und die lokale Umgebung ρ darstellen, eine Umordnung der Argument–Stacks erfolgen. Anschließend müssen die Zeiger auf die Stacks (`SpA` und `SpB`) entsprechend korrigiert werden.

Der resultierende C–Code zur Argumentübergabe an die Funktion `g` im obigen Beispiel lautet somit:

```
temp1 = SpA[1];
SpA[1] = SpA[2];
SpA[2] = temp1;

SpB[-3] = SpB[-1];
SpB -= 1;
```

(`temp1` stellt dabei eine lokale C–Variable dar. Das Argument `d#` befindet sich bereits an der richtigen Position im B–Stack.)

6.4.1.2 Der Aufruf der Funktion

Bevor die eigentliche Funktion aufgerufen werden kann, muß der `Node`–Zeiger (eine globale C–Variable) auf den die Funktion darstellenden Heapknoten verweisen. Dieser `Node`–Zeiger gestattet der Funktion, während ihrer Reduktion auf Werte, die sich im zugehörigen Heapknoten befinden (freie Variablen, Konstruktorkomponenten, Argumente) zuzugreifen.

Dieses Initialisieren des `Node`–Zeigers kann entfallen, falls der Heapknoten keine solchen Informationen enthält, wie dies z. B. für viele global definierte

Funktionen der Fall ist (Ausnahmen bilden global definierte Felder, vgl. Abschnitt 3.2.2).

Bei lokal (mit Hilfe des `let(rec)`–Ausdrucks) definierten Funktionen muß jedoch das Update–Flag berücksichtigt werden. Eine als updatable gekennzeichnete Funktion erzeugt zu Beginn ihrer Reduktion einen Update–Frame (vgl. Regel 2), der die Adresse des zugehörigen Heapknotens enthält (vgl. Abschnitt 5.4.1). Während diese Adresse bei global definierten Funktionen statisch (also bekannt) ist, muß sie bei lokal definierten Funktionen im `Node`–Zeiger übergeben werden.

Die Initialisierung des `Node`–Zeigers erfolgt vor der oben beschriebenen Argumentübergabe, da sich die Adresse des Heapknotens auf den Argument–Stacks, d. h. in der lokalen Umgebung, befinden kann.

Der eigentliche Aufruf der Funktion erfolgt im „Normalfall“ über den Funktionszeiger im zugehörigen Heapknoten (bzw. in der Informationstabelle) und zwar über den Standardeinsprung, bei dem zunächst überprüft wird, ob genügend Argumente zur Reduktion der Funktion vorhanden sind. Hierzu muß der oben beschriebene `Node`–Zeiger die Adresse des Heapknotens enthalten. Der Aufruf erfolgt dann mit dem Makro

```
ENTER( Node );
```

(Dieses Makro wird verwendet, um eine einfache Anpassung an die Laufzeitumgebung zu ermöglichen, vgl. Abschnitt 7.4.8.)

Diese Art des Funktionsaufrufs enthält zwei indirekte Speicherzugriffe

- ein Zugriff auf die Adresse der Informationstabelle über den `Node`–Zeiger
- und ein Zugriff auf die Funktionsadresse in der Informationstabelle

Erfolgen muß dieser Aufruf immer dann, wenn über die aufzurufende Funktion keine Informationen zur Verfügung stehen. Beispiel:

```
apply = {} \n {f,x} -> f {x}
```

Die anzuwendende Funktion `f` wird als Argument übernommen. Es ist weder bekannt, um welche Funktion es sich handelt, noch, ob diese Funktion tatsächlich einstellig ist.

Optimieren läßt sich der Funktionsaufruf, falls die anzuwendende Funktion (d. h. deren Definition) bekannt ist:

```
twice = {} \n {f} -> let
                        g = {f} \n {x#} -> case f {x#} of
                                                    n# -> f {n#}
                                                esac
                      in g {}
```

Bei der Anwendung der Funktion `g` sind sowohl deren Stelligkeit als auch die Namen ihrer Reduktionsfunktionen (Standardeinsprung und direkter Einsprung, vgl. Abschnitt 5.6.2) bekannt. Der Zugriff auf die Funktion muß also nicht über das Info–Table erfolgen. Vielmehr kann die Funktion über ihren Namen angesprochen werden:

```
JUMP( g_entry );
```

Wird die Funktion auf eine ihrer Stelligkeit entsprechende Anzahl von Argumenten angewandt, so kann der Test auf das Vorhandensein aller benötigten Argumente entfallen. Somit kann der direkte Einsprung aufgerufen werden:

```
JUMP( g_direct );
```

(Auch hier dient das Makro `JUMP` zur einfachen Anpassung des erzeugten C–Programms an die Laufzeitumgebung, vgl. Abschnitt 7.4.8)

6.4.2 Die Übersetzung eines `let(rec)`–Ausdrucks

Die Übersetzung eines `let(rec)`–Ausdrucks der Form

$$\begin{array}{ll} \texttt{let} & f_1 \texttt{ = } fs_1 \ \backslash\pi_1\ as_1 \texttt{ -> } b_1 \\ & \ldots \\ & f_n \texttt{ = } fs_n \ \backslash\pi_n\ as_n \texttt{ -> } b_n \\ \texttt{in } e & \end{array}$$

besteht aus zwei Teilen:

- der Erzeugung des C–Codes, der bei der Reduktion der `let(rec)`–Anweisung ausgeführt werden muß (also die Erzeugung und Initialisierung der Heapknoten für die lokal definierten Funktionen $f_1, \ldots, f_n$ und die Auswertung des Ausdrucks e)
- und der Erzeugung der Reduktionsfunktionen und der Informationstabellen für die Funktionen $f_1, \ldots, f_n$.

6.4.2.1 Der C–Code zur Reduktion eines `let(rec)`–Ausdrucks

Während der Reduktion des `let(rec)`–Ausdrucks werden zunächst die Heapknoten erzeugt, die zur Darstellung der Funktionen $f_1, \ldots, f_n$ dienen.

Zur Erzeugung der Heapknoten muß zunächst Heapspeicher beschafft werden. Die Anzahl der benötigten Heapzellen geht direkt aus den Funktionsdefinitionen hervor. So werden für jede Funktion eine Heapzelle für den Zeiger auf die Informationstabelle und je eine Heapzelle für die freien Variable fs_i der Funktion benötigt.

Die Reservierung des Heapspeichers erfolgt, indem der Zeiger auf die erste freie Heapzelle (`HP`), entsprechend der Anzahl der benötigten Heapzellen, korrigiert wird. Dabei muß auf ein Überschreiten der Heapgrenze (`HLimit`) geachtet werden, da dann eine Garbage Collection erforderlich ist.

Der C–Code zur Beschaffung des im `let(rec)`–Ausdrucks benötigten Heapspeichers sieht also wie folgt aus (dabei wird angenommen, daß insgesamt 6 Heapzellen benötigt werden; `gc` ist der Name der Funktion, die die Garbage Collection ausführt):

```
if ( HP - (6) < HLimit ) /* Platz im Heap? */
        gc(6);           /* nein, dann Garbage Collection */
HP -= 6;                 /* Speicher allokieren */
```

Direkt im Anschluß an die Reservierung des Heapspeichers wird dieser initialisiert. Dabei wird mit Hilfe des Zeigers `HP` den bereits reservierten Heapzellen ein Wert zugewiesen (z. B.):

```
HP[1] = &f_info;     /* Zeiger auf die Informationstabelle */
HP[2] = SpB[-1]; /* Freie Variable, die auf dem B-Stack */
                     /* uebergeben wurde */
```

Nach der Initialisierung der Heapknoten folgt der Code für den Rumpf (also den Ausdruck e) des `let(rec)`–Ausdrucks.

6.4.3 Die Übersetzung eines `case`–Ausdrucks

Die Auswertung eines Ausdrucks der Form `case` e `of` *alts* besteht aus der Erzeugung eines Return–Frames, zur Sicherung der Alternativen *alts* und der lokalen Umgebung ρ, gefolgt von der Auswertung des Ausdrucks e (vgl. Regel 6).

Bei der Übersetzung einer `case`–Anweisung muß also C–Code erzeugt werden, der

- die lokale Umgebung ρ sichert (diese Sicherung erfolgt nicht im Return–Frame)
- den Return–Frame erzeugt
- und den Ausdruck e auswertet.

Zusätzlich muß der Code für die Alternativen *alts* erzeugt werden.

6.4.3.1 Die Sicherung der lokalen Umgebung ρ

Die Sicherung der *verteilten* lokalen Umgebung ρ (vgl. Abschnitt 5.9) erfolgt, indem alle in den Alternativen *alts* benötigten Werte auf die Argument–Stacks gelegt werden (die Auswertung des Ausdrucks e verändert diese Stack–Elemente nicht). Dabei hängt die Sicherung dieser Werte davon ab, wo sie sich gerade befinden:

- Funktionsargumente befinden sich bereits auf den Argument–Stacks. Es muß also kein Code für deren Sicherung erzeugt werden.

- Konstruktorkomponenten müssen aus den Return–Registern auf die Stacks kopiert werden.
- Freie Variablen befinden sich im Heapknoten der gerade zu reduzierenden Funktion. Hier bieten sich zwei Möglichkeiten zur Sicherung an. Entweder kann der Zeiger auf den Heapknoten gesichert werden oder die freien Variablen im Heapknoten werden einzeln auf die Stacks gelegt. Im ersten Fall können durch die Sicherung des Zeigers gleichzeitig alle freien Variablen gesichert werden. Dies ist jedoch mit einem indirekten Zugriff auf diese Variablen in den Alternativen verbunden. Außerdem kann dieses Verfahren zu einer sehr ineffizienten Speicherausnutzung (sog. *space leaks*) führen: Wenn der Heapknoten z. B. einen Zeiger auf eine sehr lange Liste enthält, die in den Alternativen nicht benötigt wird, der Zeiger auf diesen Heapknoten jedoch aufgrund einer anderen, benötigten freien Variable gesichert wird, dann kann die Garbage–Collection den durch die Liste belegten Speicherplatz nicht freigeben, obwohl die Liste nicht mehr benötigt wird. Aus diesem Grund werden die freien Variablen einzeln auf den Stacks gesichert.

Bei dem Kopieren von Werten aus der lokalen Umgebung ρ auf die Stacks können auf den Stacks vorhandene Elemente, die nicht in den Alternativen benötigt werden, überschrieben werden. Dadurch werden ebenfalls *space leaks* (also Speicherplatz, der von der Garbage–Collection nicht freigegeben werden kann, obwohl er nicht mehr benötigt wird) vermieden. Peyton Jones [Pey92] bezeichnet dieses Überschreiben von nicht mehr benötigten Stackelementen als *stack stubbing*.

6.4.3.2 Die Erzeugung des Return–Frames

Der Return–Frame besteht aus ein oder zwei Elementen. Besitzt der in der `case`–Anweisung zu betrachtende Ausdruck e einen algebraischen Datentyp, der aus mehr als einem Konstruktor besteht, so enthält der Rahmen einen Zeiger auf eine Sprungtabelle, die die Adressen der Alternativen (d. h. der C–Funktionen, die diese Alternativen darstellen) enthält, und einen Zeiger auf die Standard–Return–Funktion `Return_from_case` (zur genauen Definition s. Abschnitt 7.1). Die Erzeugung des Return–Frames sieht dann also wie folgt aus:

```
SpB[0] = f_tab;        /* Sprungtabelle */
SpB[1] = Return_from_case;
SpB += 2;
```

(Der Name der Sprungtabelle besteht immer aus dem Namen der gerade reduzierten Funktion und dem Suffix `_tab`. Als Beispiel wurde hier der Funktionsname `f` gewählt.)

Existiert nur eine Alternative oder besitzt der Ausdruck e einen primitiven Datentyp, so besteht der Return–Frame nur aus einem Zeiger auf eine Funktion.

Diese Funktion stellt dann entweder die einzig vorhandene Alternative dar oder besteht aus einer *if-else*–Kette, die die richtige Alternative anhand des zurückgelieferten primitiven Wertes bestimmt (s. u.). Der Name dieser Funktion besteht immer aus dem Namen der gerade reduzierten Funktion und dem Suffix `_cont`:

```
SpB[0] = f_cont;
SpB += 1;
```

(Auch hier wurde als Beispiel der Funktionsname `f` gewählt.)

Im Anschluß an den Code zur Erzeugung des Return–Frame folgt der Code zur Reduktion des Ausdrucks e.

6.4.3.3 Der C–Code für die Alternativen und die Sprungtabellen

Hat der im `case`–Ausdruck zu betrachtende Ausdruck e einen algebraischen Datentyp, so wird jede Alternative zu einer eigenen C–Funktion übersetzt. Die Namen dieser C–Funktionen bestehen aus dem Namen der gerade zu übersetzenden Funktion und `_alt` gefolgt vom Konstruktortag, bei dessen Auswertung diese Alternative ausgeführt werden soll[1]. Die Adressen dieser Funktionen (also deren Namen) werden in einer Sprungtabelle zusammengefaßt, die den Namen der gerade zu übersetzenden Funktion mit dem Suffix `_tab` besitzt. Der C–Code einer Sprungtabelle (für eine `case`–Anweisung in einer Funktion `f` mit zwei Alternativen) lautet also:

```
JumpTab f_tab[] = {
        f_alt0,
        f_alt1};
```

(`JumpTab` ist dabei eine Typvereinbarung, die „Zeiger auf eine Funktion“ definiert.)

Die Auswahl der richtigen Alternative mit Hilfe einer Sprungtabelle ist nicht möglich, falls der Ausdruck e einen primitiven Typ besitzt. In diesem Fall werden alle Alternativen in eine C–Funktion übersetzt, in der dann mit Hilfe einer *if-else*–Kette die richtige Alternative ausgewählt wird. Der Name dieser Funktion besteht wieder aus dem Namen der gerade zu übersetzenden Funktion und dem Suffix `_cont`.

6.4.4 Die Übersetzung einer Konstruktoranwendung

Wie Regel 7 der operationalen Semantik zeigt, werden bei einer Konstruktoranwendung die Rückgaberegister mit dem Konstruktortag und den Konstruktorkomponenten geladen. Diese Anwendung wird in einfache Zuweisungen an die globalen C–Variablen übersetzt, die die Rückgaberegister darstellen. Diese Zuweisungen können hier unabhängig von der Konstruktorstelligkeit erfolgen, da im

[1] Ist nur eine Alternative vorhanden, so erhält diese das Suffix `_cont` (s. o.).

Gegensatz zu Peyton Jones [Pey92], hier immer genügend Rückgaberegister zur Verfügung stehen (vgl. Abschnitt 5.2.1).

Dieser Konstruktorrückgabemechanismus kann, besonders bei Konstruktoren mit großer Stelligkeit, ineffizient sein, wenn im folgenden z. B. nur eine der Konstruktorkomponenten benötigt wird. In folgenden Beispiel:

```
let fst = {} \n {x} -> case x of
                          Fuenftupel {a,b,c,d,e} -> a
                        esac
    y = {a,b,c,d,e} \n {} -> Fuenftupel {a,b,c,d,e}
in
    fst {y}
```

in dem die Variablen `a`, `b`, `c`, `d` und `e` in einem umgebenden Geltungsbereich gebunden sind, wird in der `case`–Anweisung der Funktion `fst` die Funktion `y` ausgewertet. Diese Auswertung liefert das Konstruktortag `Fuenftupel` und die Konstruktorkomponenten `a`, ..., `e` in den Rückgaberegistern zurück, obwohl in der Funktion `fst` nur eine Komponente (nämlich `a`) benötigt wird. Peyton Jones [Pey92] schlägt in diesem Fall folgenden Konstruktorrückgabemechanismus vor: Bei der Auswertung von `y` wird lediglich ein Zeiger auf den zu `y` gehörenden Heapknoten zurückgegeben. Die Funktion `fst` kann dann über diesen Zeiger indirekt auf die Konstruktorkomponenten (d. h. die freien Variablen von `y`) zugreifen. Natürlich muß der verwendete Rückgabemechanismus in den Funktionen `y` und `fst` übereinstimmen. Dies wird erreicht, indem dieser Mechanismus z. B. für jeden Datentyp einzeln festgelegt wird. Eine andere Möglichkeit besteht darin, ab einer bestimmten Stelligkeit des Konstruktors den Rückgabemechanismus zu wechseln.

Trotzdem werden hier immer das Konstruktortag und die Komponenten in den Rückgaberegistern übergeben. Eine Übergabe des Zeigers auf den Heapknoten ist nur dann sinnvoll, wenn sich die Konstruktorkomponenten (wie im oben angegebenen Beispiel) bereits in einem Heapknoten befinden. Bei einer Funktion wie der folgenden:

```
y = {} \n {a,b,c,d,e} -> Fuenftupel {a,b,c,d,e}
```

müßte sonst (evtl. unnötigerweise) erst ein Heapknoten erzeugt werden. Die Verwendung beider Übergabemechanismen für den Konstruktor `Fuenftupel` ist jedoch nicht möglich, da nicht mehr feststellbar ist, wann welcher Rückgabemechanismus benutzt werden soll.

Außerdem birgt die Übergabe des Zeigers auf den Heapknoten wieder die Gefahr von *space leaks*, da der Heapknoten Zeiger auf sehr große Datenstrukturen enthalten kann, die im folgenden garnicht benötigt werden (vgl. Abschnitt 6.4.3.1).

Bei Peyton Jones [Pey92] macht diese Optimierung durchaus Sinn, da dort Konstruktoren ab einer bestimmten Stelligkeit, aufgrund einer ungenügenden

Anzahl von (Hardware–)Rückgaberegister, sowieso im Heap erzeugt werden. Der Laufzeitgewinn durch das Sparen der Erzeugung dieser Heapknoten (vgl. Abschnitt 5.2) dürfte jedoch die hier evtl. auftretenden Laufzeitverluste bei der Rückgabe von Konstruktoren mit sehr großer Stelligkeit übersteigen.

6.4.5 Primitive Werte und Arithmetik

Primitive Werte, also Literale, Variablen von primitivem Typ und arithmetische Operationen, deren Ergbnis primitiv ist, können entweder als eigenständige STG–Funktion vorkommen, z. B.

```
f = {} \n {} -> 5
```

oder als zu betrachtender Ausdruck in einem `case`–Ausdruck:

```
case 5 of ...
```

Treten diese primitiven Werte als eingeständige Funktion auf, so werden sie zu einer C–Anweisung übersetzt, die das Rückgaberegister für primitive Werte mit dem entsprechenden Wert lädt (vgl. Regeln 12, 13 und 16). Anschließend wird dann mit der Funktion im Return–Frame auf dem B–Stack fortgefahren. Beim obersten Frame auf dem B–Stack muß es sich um einen Return–Frame handeln. Ein Update–Frame würde zum Überschreiben eines Heapknotens führen. Dies ist jedoch nicht möglich und entspricht einem Typfehler des Programms.

Funktionen, wie die oben angegebene Funktion `f`, sind jedoch recht trivial und sollten durch geeignete (STG–) Programmtransformationen (Funktionsintegration, *function inlining*) entfernt werden.

Kommen primitive Werte als zu betrachtender Ausdruck in einem `case`–Ausdruck vor, so wird das Rückgaberegister nicht geladen. Vielmehr wird direkt zu der entsprechenden Alternative verzweigt (vgl. Regel 17 und 18). Auch wenn der primitive Wert durch eine `default`–Variable gebunden wird, erfolgt die Berechnung (bei der Anwendung eines primitiven Operators) erst an der Stelle im Programm, an der dieser primitive Wert verwendet wird. So wird zum Beispiel folgender Ausdruck

```
case + {x#,y#} of
    a# -> case * {x#,4} of
        b# -> / {a#,b#}
          esac
esac
```

bei dem `x#` und `y#` in einem umgebenden Geltungsbereich gebunden sind, zu folgendem C–Code übersetzt:

```
RetRegU = (SpB[-1] + SpB[-2]) / (SpB[-1] * 4);
```

Dabei wird davon ausgegangen, daß sich die Variablen `x#` und `y#` auf dem B–Stack befinden. Und `RetRegU` bezeichnet das Rückgaberegister für primitive Werte.

Kapitel 7

Die Laufzeitumgebung

7.1 Der Return–Mechanismus

Bei der Auswertung eines `case`–Ausdrucks der Form

`case` *expr* `of` *alts*

wird vor der Auswertung von *expr* ein Return–Frame erzeugt, um danach zu einer der Alternativen aus *alts* zu gelangen. Hierzu enthält der Return–Frame einen Code–Pointer auf die Standard–Return–Funktion, die anhand des reduzierten Wertes von *expr* (der immer in den Return–Registern erzeugt wird) zu der richtigen Alternative aus *alts* verzweigt, oder, falls nur eine Alternative vorhanden ist, einen Code–Pointer auf diese Alternative.

Nach der Reduktion des Ausdrucks *expr* zu einem algebraischen bzw. primitiven Wert wird laut Semantik der Befehl *ReturnCon* bzw. *ReturnInt* ausgeführt. Diese Befehle entsprechen im zu erzeugenden C–Programm dem Makro `RETURN`[1]. Dieses Makro entfernt den obersten Return–Frame und springt zu der Funktion, deren Adresse (o. beschr. Code–Pointer) in diesem Frame enthalten ist.

Anders als in der operationalen Semantik befindet sich dieser Return–Frame jedoch nicht auf einem gesonderten Stack, sondern auf dem B–Stack. Dies bereitet jedoch keine Probleme, da dieser Frame bei der Ausführung des Makros `RETURN` stets ganz oben auf dem B–Stack liegt, also nicht durch Argumente, die auf dem B–Stack übergeben werden, abgedeckt wird. (Die einzige Ausnahme bildet ein Update–Frame, der den Return–Frame abdecken kann. Dies spielt für das Makro `RETURN` jedoch keine Rolle, da dann dieser Update–Frame ganz oben liegen muß und genauso wie ein Return–Frame verarbeitet werden kann).

Dies wird gewährleistet durch die Eigenschaft aller Funktionen die an sie übergebenen Argumente von den Argumentstacks zu entfernen. In der operationalen Semantik erfolgt dies zu Beginn jeder Funktion, indem die Argumente in die lokale Umgebung ρ eingefügt werden (Regel 3). Im zu erzeugenden C–Programm

[1] Hier wurde ein Makro gewählt, um eine einfachere Anpassung an die Laufzeitumgebung zu gewährleisten, s. Abschnitt 7.4.8

werden die Argumente erst nach der Ausführung der Funktion von den Stacks entfernt. So wird z. B. folgende Funktion

```
f = {} \n {a, b} -> <0,10> {a,b}
```

zu folgendem C–Code übersetzt[2]

```
unsigned *f_direct(void)
{
        RetReg0 = 3;
        RetReg1 = 131082;
        RetReg2 = SpA[1];
        RetReg3 = SpA[2];
        SpA += 2;

        {RETURN;}

}
```

Am Ende der Funktion, noch vor dem Makro `RETURN`, werden die auf dem A–Stack übernommenen Argumente entfernt (`SpA += 2;`).

7.2 Der Update–Mechanismus

Wird ein Ausdruck reduziert, der als updatable gekennzeichnet ist (d. h. der nach seiner Auswertung durch seinen Wert ersetzt werden soll), so wird zunächst ein Update–Frame erzeugt. Dieser Update–Frame enthält die Adresse des Heapknotens, der den Ausdruck darstellt und der mit dessen Wert überschrieben werden soll.

Dabei müssen zwei Arten von Updates unterschieden werden:

- ein Ausdruck wird durch einen algebraischen Wert (einen Konstruktor) ersetzt oder
- ein Ausdruck wird durch eine partielle Applikation ersetzt.

Im ersten Fall wird der Update (genau wie beim Return–Mechanismus) durch das Makro `RETURN` eingeleitet. Dieses Makro springt nun die Update–Funktion im Update–Frame an, da dieser den Return–Frame überdeckt.

Im zweiten Fall wird der Update durch den Standardeinsprung einer Funktion initiiert, der zunächst überprüft, ob genügend Argumente für deren Reduktion auf den Argument–Stacks vorhanden sind (vgl. Abschnitt 6.3.2). Ist dies nicht der Fall, so wird ein Update ausgeführt.

[2]RetReg0 enthält dabei die Anzahl der verwendeten Register und 131082 stellt das *erweiterte Konstruktortag* dar, s. Abschnitt 7.2

Allerdings kann nun der Update–Frame durch Argumente auf dem B–Stack überdeckt sein. Er kann jedoch trotzdem erreicht werden, da bei der Erzeugung des Frames die Argument–Stacks gesichert wurden. Der Update–Frame liegt also ganz oben im zuletzt gesicherten B–Stack, d. h. direkt unterhalb von `SBaseB` (vgl. Abschnitt 5.1.1 Die Sicherung der Argument–Stacks).

7.2.1 Das Überschreiben eines Heapknotens

Soll ein Ausdruck (d. h. der zugeh. Heapknoten) durch einen Konstruktor überschrieben werden, so muß der Konstruktor und seine Komponenten (der immer in den Return–Registern erzeugt wird) in diesen Heapknoten kopiert werden. Außerdem muß der im Heapknoten enthaltene Zeiger auf die Info–Tabelle verändert werden, da diese die Zeiger auf die Reduktionsfunktionen enthält. Eine weitere Reduktion des überschriebenen Heapknotens soll jedoch lediglich diesen Konstruktor und seine Komponenten in die Return–Register zurückschreiben.

Peyton Jones [Pey92] schlägt hierzu „Standard–Info–Tabellen" vor, die jeweils eine auf die Konstruktorstelligkeit angepaßte Reduktionsfunktion enthalten (die den Konstruktor dann in die Return–Register kopiert). Eine solche Standard–Info–Tabelle kann dann für alle Konstruktoren einer bestimmten Stelligkeit verwendet werden.

Dies bedeutet jedoch, daß das Programm so viele Standard–Info–Tabellen und Reduktionsfunktionen enthalten muß, wie es Konstruktoren mit unterschiedlicher Stelligkeit gibt. Dies ist mit ein Grund für die extrem großen ausführbaren Programme bei Peyton Jones (z. B. 700k Bytes für ein einfaches „Hello, world!"). Deshalb wird hier ein anderes Verfahren verwendet:

In die Konstruktortags wird zusätzlich die Stelligkeit des Konstruktors codiert (die unteren 8 Bit enthalten das Tag, die oberen 8 Bit enthalten die Stelligkeit)[3]. Diese *erweiterten Konstruktortags* gestatten somit die Verwendung einer einzigen Info–Tabelle für alle Konstruktoren.

Beim Überschreiben eines Heapknotens mit einer partiellen Applikation (f $x_1, \ldots, x_n$ wobei die Stelligkeit von f größer als n ist) wird ähnlich verfahren. Laut Semantik wird ein Heapknoten erzeugt, der außer den bereits vorhandenen Argumenten $x_1, \ldots, x_n$ auch die anzuwendende, aufgrund nicht genügend verhandener Argumente, nicht reduzierbare Funktion it f enthält (vgl. Regel 4),

$$(\{f, x_1, \ldots, x_n\} \setminus n \; \{\} \texttt{ -> } f\,\{x_1, \ldots, x_n\})\;(a{:}v_1, \ldots, v_n)$$

wobei a die Heapadresse der Funktion f und $v_1, \ldots, v_n$ die Werte der Argumente $x_1, \ldots, x_n$ beschreiben. Bei einer Reduktion dieses Knotens müssen die Argumente, die nun zu freien Variablen geworden sind, zurück auf die Argument–Stacks kopiert werden und die Funktion f darauf angewandt werden.

[3]Genau genommen werden die Tags durch 32 Bit–Worte dargestellt. Die übrigen 16 Bit dienen zur Codierung der Anzahl der boxed und der unboxed Komponenten des Konstruktors, um die Garbage Collection zu vereinfachen (s. Abschnitt 7.3)

Um nicht für jede mögliche Anzahl vorhandener Argumente (bzw. freier Variablen) eine eigene Reduktionsfunktion zu verwenden (bei einer n-stelligen Funktion werden (n - 1) Reduktionsfunktionen benötigt), wird wieder eine Standard–Reduktionsfunktion verwendet, die bei allen partiellen Applikationen verwendet werden kann und die Anzahl der vorhandenen Argumente anhand einer Markierung des Heapknotens erkennt.

Die Info–Tabelle kommt für diese Markierung jedoch nicht in Frage, da auch diese Tabelle für alle partiellen Applikationen verwendet werden soll. Um die Heapstruktur (Zeiger auf die Info–Tabelle, gefolgt von den freien Variablen) beizubehalten wird diese Markierung von den Beginn des Heapknotens gesetzt[4].

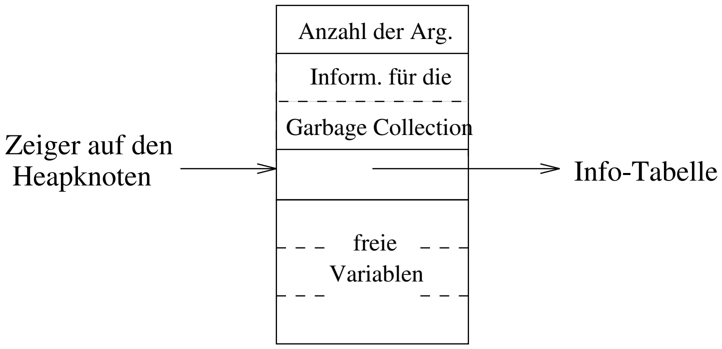


Abbildung 7.1: *Die Heapknotenstruktur partieller Applikationen*

Somit reichen auch bei Updates mit partiellen Applikationen eine einzige Info–Tabelle und eine einzige Reduktionsfunktion aus.

7.3 Die Garbage–Collection

Eine Garbage Collection wird immer dann gestartet, wenn der Heapspeicher nicht mehr ausreicht, um einen neuen Heapknoten anzulegen. Dann müssen die noch benötigten Heapknoten bestimmt werden und alle anderen Knoten aus dem Heap entfernt werden. Hierzu gibt es viele Verfahren: Mark-Scan-, Two-Space-, Generational–Garbage Collection (s. hierzu z. B. [Co81], [Ap89]). An dieser Stelle soll jedoch keines dieser Verfahren beschrieben werden. Vielmehr soll hier kurz erklärt werden, worauf bei einer Garbage Collection zu achten ist.

7.3.1 Die Bestimmung der noch benötigten Heapknoten

Um die im weiteren Programmverlauf noch benötigten Heapknoten zu identifizieren, müssen deren Heapadressen bestimmt werden. Diese Adressen befinden

[4]Dort werden auch noch weitere Informationen für die Garbage Collection gespeichert

sich

- auf dem A–Stack. Der A–Stack enthält ausschließlich Adressen von Heapknoten.
- in den Return–Registern. Hier kann sich ein Konstruktor befinden, der im weiteren Programmverlauf noch benötigt wird und dessen Komponenten Zeiger auf Heapknoten sein können.
- in den Update–Frames auf dem B–Stack. Die Update–Frames enthalten die Adressen der später zu überschreibenden Heapknoten.

Die Adressen auf dem A–Stack können alle ohne weiteres bearbeitet werden. Bei den Return–Registern hingegen stellt sich die Frage, welche Komponenten des Konstruktors Zeiger auf Heapknoten sind und welche primitive Werte sind, die keinesfalls als Zeiger interpretiert werden dürfen.

Zur Klärung dieser Frage dient das schon erwähnte erweiterte Konstruktortag, das auch die Anzahl der Zeiger und der primitiven Werte unter den Komponenten angibt.

Anzahl der primitiven Werte	Anzahl der Zeiger-Werte	Stelligkeit des Konstruktors	Konstruktortag
8 Bit	8 Bit	8 Bit	8 Bit

Abbildung 7.2: *Der Aufbau des erweiterten Konstruktortags*

Da jedoch die Reihenfolge der Komponenten nicht aus diesen Informationen hervorgeht, werden Konstruktoren so aufgebaut, daß ihre Komponenten, die Zeiger beschreiben immer vor den Komponenten stehen, die primitive Werte beschreiben. Somit bezieht sich die im Konstruktor enthaltene Anzahl der „Zeigerkomponenten“ immer auf die ersten Komponenten des Konstruktors (eine beliebige Reihenfolge der Komponenten ist zwar möglich, macht jedoch die Garbage Collection wesentlich komplizierter).

Die Zeiger in den Update–Frames besitzen relativ zum Anfang des Frames immer die gleiche Adresse. Ein Problem ist jedoch die Update–Frames im B–Stack zu finden, da sich zwischen diesen Frames beliebig viele Return–Frames und Argumente befinden können.

Wie schon beschrieben, befindet sich der oberste Update–Frame direkt unterhalb von `SBaseB` (also ganz oben im zuletzt gesicherten Stack). Da bei jeder Erzeugung eines Update–Frames die Argument–Stacks gesichert werden und später wieder hergestellt werden müssen, enthalten die Update–Frames auch die Basiszeiger (`SBaseA` bzw. `SBaseB`) der zuletzt gesicherten Stacks. Mit Hilfe dieser Zeiger kann die Garbage Collection dann den nächsten Update–Frame lokalisieren (Abb. 7.3.1.

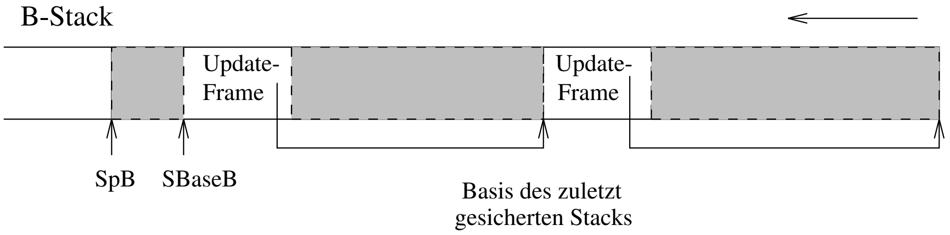


Abbildung 7.3: *Die „Verkettung" der Update–Frames*

7.4 Tail Call Optimierung und Direktsprungsimulation

7.4.1 Tail Calls

Von einem *Tail Call* spricht man, wenn der Wert einer Funktion f durch den Aufruf einer anderen Funktion g definiert ist (d. h. wenn die Funktion g am Ende der Funktion f aufgerufen wird). Handelt es sich bei der aufgerufenen Funktion um die Funktion f selbst, so spricht man von *Tail Call–Rekursion*. Der Wert der Funktion f steht also schon fest, sobald die Funktion g (bzw. der rekursive Aufruf der Funktion f) ausgewertet ist. Ein Zurückkehren zur Funktion f erscheint unnötig, da diese bereits fertig berechnet ist.

7.4.2 Tail Call Optimierung

Dieser unnötige Rücksprung zur Funktion f wird vermieden, indem die Funktion g nach ihrer Reduktion zur der Stelle im Programm zurück springt, an der die Funktion f angewandt wurde. Diese Optimierung bezeichnet man als „Tail Call Optimierung" (s. Abb. 7.4).

7.4.3 Voraussetzung für Tail Call Optimierung

Eine Tail Call Optimierung ist nur dann zu realisieren, wenn die Funktion g nicht zur Funktion f zurückkehren muß, wie dies bei einem Unterprogrammaufruf der Funktion g der Fall ist. Eine Ausführung der Funktion g mittels Unterprogrammaufruf führt zur Speicherung einer Rücksprungadresse (bzw. eines *Activation Records*), die der Funktion g den Rücksprung zur Funktion f ermöglicht. Wird dieser Rücksprung aufgrund der Tail Call Optimierung nicht ausgeführt, bleibt diese Rücksprungadresse auf dem Systemstack liegen, was bei wiederholten Tail Calls zu einem Stack Overflow führen kann. Die Funktion g muß also direkt angesprungen werden, damit kein Activation Record erzeugt wird.

7.4.4 Funktionsanwendung in C

Funktionen in C bieten die Möglichkeit Anweisungsfolgen zu einem Block zusammenzufügen und mit einem Namen zu versehen, mit dessen Hilfe diese Anweisungsfolgen angesprungen werden können. Diese Funktionsnamen werden als Zeiger auf die Funktion betrachtet und können wie gewöhnliche Zeiger beliebig manipuliert werden. So kann der Name einer Funktion als Argument übergeben oder im Speicher abgelegt werden. Da C nur globale Funktionen erlaubt[5], stehen diese Funktionsnamen im gesamten Programm zur Verfügung[6].

Der Aufruf von C–Funktionen erfolgt jedoch nach dem Unterprogrammkonzept. D. h. bei jeder Funktionsanwendung wird ein Activation Record auf dem Systemstack erzeugt, wodurch eine Tail Call Optimierung verhindert wird.

Eine Möglichkeit Anweisungsfolgen direkt anzuspringen, stellt C in Form von Marken zur Verfügung (vgl. Kernighan, Ritchie [KR83]). Marken (oder Code–Labels) können vor jeder beliebigen C–Anweisung definiert werden. Mit Hilfe der `goto`–Anweisung kann dann direkt zu einer dieser Marken gesprungen werden. Der Geltungsbereich einer Marke erstreckt sich jedoch nur über den Funktionsblock, in dem die Marke definiert wurde, was den Direktsprung zu einer Marke, die in einer anderen Funktion definiert wurde, verhindert. Auch die hieraus offensichtlich folgende Darstellung eines Programms als eine einzige C–Funktion (damit alle Direktsprünge innerhalb dieser einzigen Funktion erfolgen und somit mit Marken realisiert werden können), erweist sich als unmöglich, da Marken ausschließlich in `goto`–Anweisungen verwendet werden dürfen und nicht als Werte abgespeichert werden können, wodurch die Übergabe von Funktionen verhindert wird[7].

Trotzdem kann der Label–Mechanismus in bestimmten Fällen zur Tail Call Optimierung herangezogen werden. Tail–rekursive Funktionen, also Funktionen, die sich selbst aufrufen, können optimiert werden, indem vor der ersten Anweisung der Funktion eine Marke definiert wird und der rekursive Aufruf durch eine `goto`–Anweisung ersetzt wird, die diese Marke anspringt.

Eine andere Möglichkeit innerhalb von C direkte Sprünge zu realisieren, besteht in der Ausnutzung einer Assembler–Schnittstelle. Viele C Compiler erlauben das Einfügen von Assembler–Code in C Programme. Dies ist jedoch stark maschinenabhängig und führt zu nicht portablem Code.

[5] Einige Compiler (wie z. B. der GNU–C–Compiler) erlauben auch die Definition lokaler Funktionen.

[6] Dies hat auch einen unerwünschten Nebeneffekt. So stehen lokal definierte STG–Funktionen durch die Abbildung auf globale C–Funktionen nun global zur Verfügung.

[7] Auch hier stellt der GNU–C–Compiler eine Ausnahme dar.

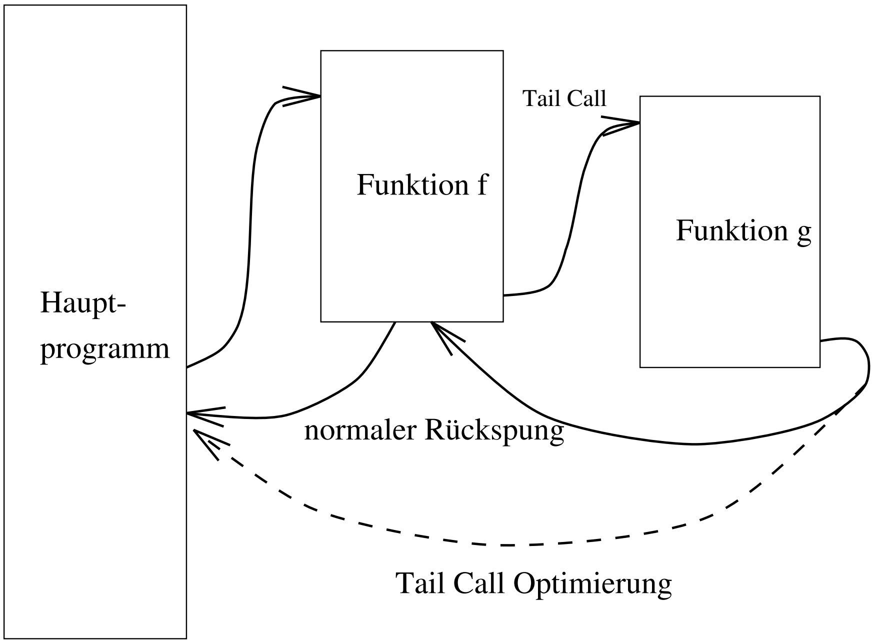


Abbildung 7.4: *Tail Call Optimierung*

7.4.5 Direktsprungsimulation

Um trotzdem eine Tail Call Optimierung in einem C Programm vornehmen zu können, wird der Direktsprung zu einer Funktion simuliert. Hierzu wird ein Mechanismus benutzt, der schon von Steele in seinem Rabbit Compiler für Scheme [Ste78] benutzt wurde und mit *UUO Handler* bezeichnet wird. Dieser UUO Handler findet ebenfalls im SML Compiler von Tarditi, Acharya & Lee [TLA91] und im STG Compiler von Peyton Jones [Pey92] Verwendung.

Der UUO Handler ist ein kleiner „Ein–Zeilen"–Interpreter, der in einer Endlosschleife läuft:

```
while (TRUE) { cont = (*cont)(); }
```

Die Variable `cont` ist als Zeiger auf Funktionen definiert und wird vor dem Starten des Interpreters mit der Adresse der zuerst auszuführenen Funktion initialisiert. Durch den Ausdruck `(*cont)()` wird die Funktion, deren Adresse `cont` enthält, ausgeführt. Will nun eine Funktion eine andere Funktion direkt anspringen, so erfolgt dies, indem sie die Adresse dieser Funktion an den Interpreter zurück gibt, der sie dann (als Unterprogramm) aufruft (s. Abb. 7.5). Dadurch wird jeweils nur die gerade ausgeführte Funktion als Unterprogramm ausgeführt und nur deren Activation Record auf dem Stack aufbewahrt. Dieser Record wird beim Rücksprung zum Interpreter wieder vom Stack entfernt, womit ein Überlaufen des Stacks vermieden wird.

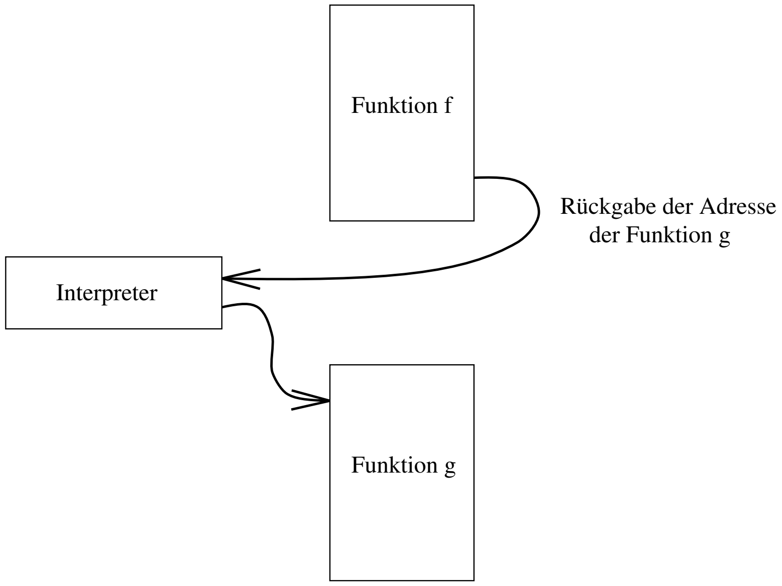


Abbildung 7.5: Direktsprungsimulation

Diese Direktsprungsimulation kann natürlich durch echte Direktsprünge (mittels Marken und `goto`–Anweisung) bei tail–rekursiven Funktionen ergänzt werden.

Bei der Verwendung dieses Interpreters müssen nun allerdings Rücksprungadressen für eventuell doch notwendige Rücksprünge selbst verwaltet werden.

7.4.6 Nachteile der Direktsprungsimulation

Die Simulation von direkten Sprüngen in C–Programmen ist sehr teuer. Sie besteht, für jeden Direktsprung, aus einem Rücksprung zum Interpreter und einem Sprung zu der direkt anzuspringenden Funktion. Jeder durch die Tail Call Optimierung „gesparte“ Rücksprung wird also durch einen zusätzlichen Rücksprung (zum Interpreter) beim Aufruf der Funktion neutralisiert. Damit ist die Anzahl der Funktionsaufrufe und Rücksprünge die gleiche wie beim Unterprogrammaufruf der Funktion. Ist der Aufruf einer Funktion kein Tail Call, d.h. nach dem Funktionsaufruf muß zu der aufrufenden Funktion zurückgekehrt werden[8], so werden aus einem Funktionsaufruf und einem (nicht “wegzuoptimierenden“) Rücksprung, der ebenfalls über den Interpreter läuft, durch die Direktsprungsimulation zwei Funktionsaufrufe und zwei Rücksprünge. In diesem Fall ist die Direktsprungsimulation sogar deutlich schlechter als das Unterprogrammkonzept.

[8]solche Funktionsaufrufe sind im C–Programm durchaus sinnvoll, z.B. bei primitiven `case`–Anweisungen, die nach der Auswertung des zu betrachtenden Ausdrucks ein Pattern Matching durchführen

Auch Optimierungen, wie z.B. Funktionsintegration und Tail Call Optimierung bei rekursiven Funktionen (s. u.), die dazu dienen die Anzahl der „teueren“ Direktsprünge zu reduzieren, verändern nichts an diesem Nachteil des Interpreters, da diese Optimierungen auch im Unterprogrammkonzept angewandt werden können und auch hier zu einer Reduzierung der Funktionsaufrufe führt.

Ein weiterer Nachteil des UUO Handlers ist eine starke Fragmentierung der „funktionalen“ Funktionen, bei der Abbildung auf C Funktionen. So werden z.B. die Alternativen einer `case`–Anweisung zu einzelnen C–Funktionen transformiert:

```
f = case g {x, y} of
        <pat1> -> <alt1>
            ...
        <patn> -> <altn>
```

Die Auswertung der Funktion g, angewandt auf die Argumente x und y, erfolgt, indem die Adresse der Funktion g an den Interpreter zurückgegeben wird. Damit wird diese C–Funktion verlassen. Um nach der Auswertung von g zu einer der Alternativen springen zu können, muß jede dieser Alternativen zu einer eigenen C–Funktion transformiert werden. Diese Fragmentierung macht eine Tail Call Optimierung von tail–rekursiven Funktionen mit dem Label–Mechanismus (s. o.), unmöglich, da der Beginn der Funktion und der rekursive Aufruf in verschiedene C–Funktionen abgebildet werden. Ein Aufruf der Funktion g als Unterprogramm führt zu einer Rückkehr zur Funktion f und macht so die Darstellung der Alternativen als eigene C–Funktionen unnötig[9].

Bei der Erzeugung von reinem C–Code empfiehlt es sich daher, wie auch folgender Zeitvergleich zeigt, das Unterprogrammkonzept zu Gunsten der Ausführungszeit, dem UUO Handler (oder „apply-like procedure“, wie dieser Interpreter von Tarditi, Lee und Acharya im SML Compiler genannt wird) vorzuziehen.

7.4.7 Laufzeitvergleiche

Tabelle 7.4.7 zeigt einen Zeitvergleich zwischen der Direktsprungsimulation und dem Unterprogrammkonzept. Zum Vergleich wurden herangezogen (s. Anhang):

Primes: Das Sieb des Eratosthenes. Die zehntausendste Primzahl wurde berechnet und ausgegeben.

Takeuchi: Die Takeuchi–Funktion mit den Startwerten 24 16 8.

Die erste Zeile gibt die Ausführungszeiten (user und system) an, bei der Verwendung des UUO Handlers, also mit Tail Call Optimierung. Tail–rekursive Funktionen benutzen ebenfalls den UUO Handler für den rekursiven Aufruf. Die

[9]Trotzdem werden auch hier die Alternativen zumindest der algebraischen `case`–Anweisungen durch eigene C–Funktionen dargestellt — s. Pattern Matching.

	Takeuchi	Primes
Direktsprungsimulation mit Tail Call Optimierung	15.82 0.08	12.80 13.56
Unterprogrammaufruf ohne Tail Call Optimierung	13.89 0.10	11.09 12.73
Unterprogrammaufruf mit partieller Tail Call Optimierung	12.95 0.08	9.94 13.30

Tabelle 7.1: *Zeitvergleiche bei der Verwendung der Direktsprungsimulation und des Unterprogrammkonzepts*

letzen beiden Zeilen geben die Zeiten an, bei der Verwendung des Unterprogrammkonzepts. Dabei wurde im ersten Fall auf jegliche Tail Call Optimierung verzichtet, während im zweiten Fall tail–rekursive Funktionen mit dem Label–Mechanismus von C optimiert wurden („partielle Tail Call Optimierung“).

Eine Optimierung der tail–rekursiven Funktionen mittels des Label–Mechanismuses, ist in den Versionen, die den UUO Handler benutzen nicht möglich, da in beiden Beispielprogrammen, aufgrund der oben erwähnten Fragmentierung, der Beginn der Funktion und der rekursive Aufruf sich nicht in einer C–Funktion befinden. Eine Verwendung des UUO Handlers schließt eine solche Optimierung nicht gänzlich aus. Jedoch wird die Anzahl der tail–rekursiven Funktionen, die so optimiert werden können, drastisch reduziert.

7.4.8 Direktsprünge in Assembler

Durch die Assembler–Schnittstelle, die die meisten C Compiler bieten, können tatsächlich direkte Sprünge ausgeführt werden. Dies ist allerdings nicht unproblematisch, vor allem, wenn man gleichzeitig lokale Variablen benutzt oder Argumente an Funktionen übergibt (vgl. Miranda [Mir91]). C–Compiler erzeugen zu Beginn jeder Funktion Code, der den Systemstack sichert und am Ende jeder Funktion Code, der den Systemstack wieder herstellt. Wird die Funktion nun mit einem Direktsprung verlassen, so wird der Code zur Wiederherstellung des Systemstacks am Ende der Funktion nicht ausgeführt, d. h. lokale Variablen oder Argumente der Funktion bleiben auf dem Systemstack liegen und können zu einem Stack Overflow führen.

Eine mögliche Lösung diese Problems wäre ein Direktsprung, der die Sicherung des Systemstacks zu Beginn jeder Funktion überspringt. Dabei müßte jedoch auf lokale Variablen und Funktionsargumente verzichtet werden. Versuche haben jedoch gezeigt, daß der C–Compiler bei bestimmten Optimierungen den Systemstack zur Zwischenspeicherung von Werten verwendet, wodurch wiederum ein Stack Overflow auftreten kann. Deshalb wurde folgende Lösung gewählt: Bevor eine Funktion durch einen Direktsprung verlassen wird, wird der Systemstack wiederhergestellt. Somit können auch lokale Variablen und Argumente verwendet werden.

Diese Lösung ist jedoch stark maschinenabhängig. Deshalb werden alle Sprünge im C–Programm durch Makros realisiert:

`JUMP (f)` springt die Funktion `f` direkt an

`Enter (a)` springt die Reduktionsfunktion (den Standardeinsprung) eines Ausdrucks mit Heapadresse a an

`RETURN` springt die Funktion im obersten Return– bzw. Update–Frame an.

Diese Makros werden in einem C–Headerfile definiert und können (ohne Veränderung eines bereits erzeugten C–Programms) an die jeweils verwendete Maschine oder den UUO Handler angepaßt werden.

Kapitel 8

Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde ein Übersetzer für die funktionale STG–Sprache entwickelt. Dieser Übersetzer wurde in Gofer, einer nicht-strikten, funktionalen Sprache, die weitestgehend der Sprache Haskell [Hud⁺92] entspricht, entwickelt und umfaßt ca. 100k Bytes Gofer–Code. Die zur Übersetzung eines erzeugten C–Programms benötigte Laufzeitumgebung (Garbage–Collection, Ausgabefunktion, ...) besteht aus ca. 40k Bytes C–Code.

Der Übersetzer verwendet alle in dieser Arbeit beschriebenen Optimierungen, von denen die wichtigsten in Tabelle 8.1 zusammengefaßt sind.

Als abschließender Test und zum Vergleich des hier implementierten Übersetzers mit anderen Übersetzern, wurde das „Pseudoknot–Programm" übersetzt. Dieses Programm löst ein Problem aus der Molekularbiologie (max. Abstand zwischen Atomen in einem Teil eines Nukleinsäuremoleküls) mit Hilfe eines Backtracking-Algorithmuses. In [Har⁺94] dient dieses Programm als Benchmark, zum Vergleich von über 20 funktionalen Übersetzern. Um dabei Laufzeitunterschiede, aufgrund unterschiedlicher Maschinen, auszuschließen, werden die Laufzeiten relativ zur Laufzeit eines vergleichbaren C–Programms[1] bestimmt. Diese relativen Laufzeiten werden in der Einheit „Pseudoknot" angegeben:

$$\text{Pseudoknots} = \frac{\text{Ausführungszeit des C–Programms} \times 100}{\text{Ausführungszeit des vom Übersetzer erzeugten Programms}}$$

D. h. ein Programm, das mit 100 Pseudoknots läuft, benötigt die gleiche Zeit wie das vergleichbare C–Programm; 50 Pseudoknots bedeuten, daß das erzeugte Programm die doppelte Zeit des C–Programms benötigt.

Der hier implementierte Übersetzer erreicht, bei der Verwendung des gleichen C–Compilers wie in [Har⁺94] (GNU–GCC, Version 2.5.8), 60 Pseudoknots (zum Vergleich: Der Glasgow Haskell Compiler, der ebenfalls auf der abstrakten STG–Maschine von Peyton Jones basiert, erreicht laut [Har⁺94] 69,5 Pseudoknots).

[1] Dieses C–Programm wird zusammen mit einigen funktionalen Versionen des Pseudoknot–Programms per FTP zur Verfügung gestellt.

Die Verwendung einer neueren Version des GNU–C–Compilers zeigt jedoch, wie sehr die Ausführungsgeschwindigkeit der Programme von den Optimierungen des verwendeten C–Compilers abhängen: Bei der Verwendung der Version 2.6.2 des GNU–Compilers ergibt sich für das hier übersetzte Pseudoknot–Programm nur noch eine relative Geschwindigkeit von knapp 28 Pseudoknots.

Anhang A beschreibt detailiert die Ergebnisse dieses Benchmark–Tests.

8.1 Ausblick

Im Hinblick auf eine parallele Implementierung erscheint die abstrakte STG–Maschine nicht uneingeschränkt geeignet. Im Gegensatz zur G–Maschine besitzt sie keine Code–Komponente (einen Evaluierer), die die Auswertung der im Heap gehaltenen Datenstrukturen steuert. Vielmehr ist in der STG–Maschine jeder Heapknoten für seine Reduktion „selbst verantwortlich". Heirzu enthält jeder Knoten den Code, (bzw. einen Zeiger auf diesen Code), der seine Reduktion durchführt.

Für einen Multiprozessorumgebung mit shared memory ist diese Struktur ideal. Soll der Übersetzer jedoch mehrere, durch ein Netzwerk verbundene Rechner verwenden, so stellt diese Struktur eher ein Hindernis dar, da außer den Datenstrukturen im Heap auch Programm–Code verteilt werden muß.

Optimierung gegenüber Peyton Jones	Effekt
Unboxed values werden in der Syntax explizit mit einem # gekennzeichnet.	Dies gestattet die Definition polymorpher Funktionen auf unboxed values und die Definition wesentlich effizienterer Datenstrukturen.
Die Anzahl der verwendeten Return–Register (dargestellt durch globale C–Variablen, die, soweit möglich, auf Hardwareregister abgebildet werden) wird der Anzahl der im Programm benötigten Return–Register angepaßt.	Rückgabewerte können so immer in den Return–Registern übergeben werden und müssen nie im Heap erzeugt werden. Dies gestattet eine wesentlich schnellere Rückgabe und einen schnelleren Zugriff auf diese Werte. Dies hat zur Folge, daß nur eine Update–Funktion benötigt wird, was wiederum Sprungtabellen in den Update–Frames unnötig macht.
Update– und Return–Frames enthalten einen Funktionszeiger zur Unterscheidung zwischen diesen beiden Frame–Arten.	Einzelne Alternativen und die Update–Funktion können direkt angesprungen werden. Dadurch vereinfacht sich der gesamte Update–Mechanismus, der bei Peyton Jones extrem kompliziert ist.
Heapknoten werden mit „typgerechter" Größe erzeugt.	Dies ermöglicht es die Heapknoten bei einem Update mit ihrem Ergebnis zu überschreiben, anstatt neue Heapknoten anzulegen. Somit wird Laufzeit und Heapspeicher gespart.
Die Konstruktortags enthalten zusätzliche Informationen über die Anzahl und die Art (boxed oder unboxed) der Konstruktorkomponenten.	Der C–Code der bei der Erzeugung (Update) eines Konstruktors im Heap benötigt wird, ist unabhängig von der Art und der Anzahl der Konstruktorkomponenten und kann so für alle Konstruktoren verwendet werden. Dies reduziert die Größe des ausführbaren Programms.

Tabelle 8.1: *Unterschiede zur Implementierung von Peyton Jones*

Anhang A

Laufzeitvergleich

Das Pseudoknot–Programm, ein Backtracking–Algorithmus zur Lösung eines Problems aus der Molekularbiologie (s. [Har$^+$94]), benötigt sehr viele Fließkommaoperationen, wie Tabelle A.1 zeigt[1].

Fließkommaoperationen		Wurzel- und trigonometrische Funktionen	
$\times$	3567672		
$+$	2798571	$\sqrt{}$	69600
$>, \leq, <$	129656	arctan	40184
$-$	330058	cos	40184
/	40184	sin	40184
gesamt	6866141	gesamt	190152

Tabelle A.1: *Fließkommaoperationen im Pseudoknot–Programm*

Es besteht aus ca. 6500 Zeilen STG–Code. Der Übersetzer erzeugt daraus ca. 26000 Zeilen C–Code (davon enthalten ca. 19500 Zeilen Funktionsprototypen und Info–Tabellen).

Bei der Ausführung benötigt das Programm 4176769 Heapzellen und führt 281418 Updates durch, von denen 17671 den alten Heapknoten überschreiben (Updates in place).

Die Haskell–Version dieses Programms wurde von Hand in STG–Code übersetzt, da kein Front-End zur Transformation in STG–Code zur Verfügung stand. Dadurch ist nicht sichergestellt, daß es sich um eine „optimale“ Version des STG–Programms handelt (die korrekte Bestimmung z. B. der Update-Flags oder der Größe der Heapknoten für Updates 'in place' ist in einem solch großen Programm von Hand nur sehr schwer möglich). Außerdem verwendet der Übersetzer keinen Zwischencodeoptimierer (dies hätte den Rahmen dieser Diplomarbeit gesprengt). Trotzdem liefert der Übersetzer gute Ergebnisse (60 Pseudoknots), die

[1] Diese Angaben wurden [Har$^+$94] entnommen.

nur knapp unterhalb des in Glasgow entwickelten, auf der STG–Maschine basierenden, Haskell–Compilers (69,5 Pseudoknots laut [Har+94], ebenfalls mit gcc Version 2.5.8 übersetzt) liegt, wie Tabelle A.2 zeigt[2]. Die Übersetzung der glei-

GCC Version 2.5.8	Laufzeit user + system	Pseudoknots
C–Programm	14.39 + 865.53	100 + 100
funktionales Programm mit Ein-Zeilen-Interpreter	31.55 + 829.07	45.6 + 104.4
funktionales Programm mit Direktsprung	23.65 + 791.66	60.8 + 109.3

Tabelle A.2: *Laufzeitvergleich bei Verwendung GNU–C–Compilers Version 2.5.8*

chen Programme mit dem GNU–Compiler der Version 2.6.2 zeigt jedoch, wie sehr diese Laufzeiten vom jeweils verwendeten C–Compiler abhängen. Bei der Verwendung dieser Compiler–Version erreicht das funktionale Programm mit Direktsprung, gegenüber dem C–Programm nur noch 27.6 Pseudoknots (Tabelle A.3)[3].

GCC Version 2.6.2	Laufzeit user + system	Pseudoknots
C–Programm	3.35 + 0.19	100 + 100
funktionales Programm mit Direktsprung	12.14 + 0.42	27.6 + 45.2

Tabelle A.3: *Laufzeitvergleich bei Verwendung GNU–C–Compilers Version 2.6.2*

[2]Diese Laufzeiten wurden auf einem PC mit Intel 80386 Prozessor und 8 MB Hauptspeicher ohne Floating-Point-Coprozessor ermittelt. Dies führt zu solch hohen Systemzeiten.

[3]Diese Laufzeiten wurden auf einem PC mit Intel 80486 Prozessor und 16 MB Hauptspeicher ermittelt.

Anhang B

Beispielprogramme

Folgende STG–Programme wurden wurden in dieser Arbeit für Laufzeitvergleiche verwendet (dabei wurden Konstruktoren in „lesbarer“ Form angegeben):

B.1 Die Takeuchi–Funktion

Die bekannte Takeuchi–Funktion

$$\text{tak (x, y, z)} = \text{tak (} \begin{array}{l} \text{tak (x-1, y, z),} \\ \text{tak (y-1, z, x),} \\ \text{tak (z-1, x, y))} \end{array}$$

arbeitet (in STG–Syntax übersetzt) ausschließlich mit primitiven Werten (unboxed values) und benötigt so keinen Heapspeicher. Ein Aufruf der Funktion mit den Eingabewerten x = 24, y = 16 und z = 8 führt zu ca. 1.5 Millionen rekursiven Aufrufen der Funktion. Der STG–Code für diese Funktion sieht wie folgt aus:

```
_tak = {} \n {x#, y#, z#} -> case <= {x#, y#} of
        0 -> case - {x#, 1} of
          x1# -> case _tak {x1#, y#, z#} of
            tak1# -> case - {y#, 1} of
              y1# -> case _tak {y1#, z#, x#} of
                tak2# -> case - {z#, 1} of
                  z1# -> case _tak {z1#, x#, y#} of
                    tak3# -> _tak {tak1#, tak2#, tak3#}
                    esac
                  esac
                esac
              esac
            esac
          esac
        n# -> z#
```

```
        esac

main = {} \n {} -> case _tak {24, 16, 8} of
                             erg# -> <0,1> {erg#}
                        esac
```

B.2 Primzahlen

Folgendes STG–Programm berechnet die 100000. Primzahl mit Hilfe des Siebs des Eratosthenes. Diese Programm benötigt sehr viel Heapspeicher, da alle berechneten Primzahlen in Form einer Liste im Speicher gehalten werden müssen.

```
_sieve2 = {} \n {p#, dist#, list} -> case + {p#, dist#} of
                n# -> case - {6, dist#} of
                        m# -> _sieve {n#, m#, list}
                        esac
                esac


_isPrime = {} \n {list, to_test#, wurzel#} -> case list of {2}
        Cons {he,tail} -> case he of {1}
            Int {h#} -> case > {h#, wurzel#} of
                0 -> case % {to_test#, h#} of
                    m# -> case == {m#, 0} of
                        0 -> _isPrime {tail,to_test#,wurzel#}
                        w# -> False
                        esac
                    esac
                n# -> True
                esac
            esac
        esac

_sieve = {} \n {p#, dist#, list} -> case sqrt {p#} of
        wurz# -> case _isPrime {list, p#, wurz#} of {2}
            True -> let prim = {p#} \n {} -> Int {p#}
                        next = {list, p#,dist#} \u {}
                                   -> _sieve2 {p#, dist#, list}
                    in
                        Cons {prim, next}
            False -> case + {p#, dist#} of
                n# -> case - {6, dist#} of
                    m# -> _sieve {n#, m#, list}
```

```
                    esac
                esac
            esac
        esac

main = {} \n {} -> letrec
           x = {z}   \u {} -> _sieve {7, 4, z}
           y = {}    \n {} -> Int {5}
           z = {y,x} \n {} -> Cons {y,x}
        in
           _take {10000,z}

_take = {} \n {n#, list} -> case list of {2}
        Cons {head, tail} -> case n# of
            1 -> head {}
            x# -> case - {n#, 1} of
                 y# -> _take {y#, tail}
                 esac
            esac
        esac
```

B.3 Das einmillionste Element der Liste der natürlichen Zahlen

Das folgende Programm berechnet das einmillionste Element der Liste der natürlichen Zahlen.

```
main = {} \n {} -> letrec
                        x = {y}   \u {} -> next {y}
                        y = {}    \n {} -> Int {1}
                        z = {x,y} \n {} -> Cons {y,x}
                        in
                                _take {1000000,z}

_take = {} \n {n#, list} -> case list of {2}
                Cons {head, tail} -> case n# of
                        1 -> head {}
                        x# -> case - {n#, 1} of
                                y# -> _take {y#, tail}
                                esac
                        esac
                esac
```

```
next = {} \n {next} -> case next of {1}
                        Int {n#} -> case + {n#,1} of
                                m# -> letrec
                                        a = {m#} \n {} -> Int {m#}
                                        b = {a} \u {} -> next {a}
                                      in Cons {a,b}
                                esac
                        esac
```

Literaturverzeichnis

[Ap89] Andrew W. Appel: *Simple Generational Garbage Collection and Fast Allocation*, in Software–Practice and Exp., Vol. 19(2), S. 171–183, Februar 1989.

[Aug84] L. Augustsson: *A compiler for Lazy ML*, in Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin, S. 218-227, August 1984.

[BELP87] T. Brus, M. van Eekelen, M. van Leer, M. J. Plasmeijer: *Clean — a Language for Functional Graph Rewriting*, Proc. of the third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, Lecture Notes on Computer Science 274, S. 346–384, Springer Verlag.

[BMOR87] W. Burton, D. Maurer, H. Oberhauser, R. Wilhelm: *A Space–Efficient Optimization of Call-By-Need*, in IEEE Transactions on Software Engineering **13**(6), S. 636–642, 1987.

[Co81] Jacques Cohen: *Garbage Collection of Linked Data Structures*, in Computing Surveys, Vol. 13, No. 3, September 1981.

[ENPS90] M. van Eekelen, E. Nöcker, M. J. Plasmeijer, J. Smetsers: *Concurrent Clean, version 0.6*, Technical Report 90-21, University of Nijmegen, Dezember 1990.

[FW87] J. Fairbairn, S. C. Wray: *TIM. A Simple, Lazy Abstract Machine to Execute Supercombinators*, in Proc. Functional Programming Languages ans Computer Architecture, LNCS 274, S. 34–35, Springer, 1987.

[Har+94] Pieter H. Hartel et al.: *Pseudoknot: a Float-Intensive Benchmark for Functional Compilers (Draft)*, 11. September 1994.

[Hud+92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, John Peterson: *Report on the programming language Haskell, Version 1.2*, SIGPLAN Notices 27, Mai 1992.

[Joh84] T. Johnsson: *Efficient compilation of lazy evaluation*, in Proceedings of the ACM Conference on Compiler Construction, Montreal, S. 58–69, Juni 1984.

[KR83] Brian W. Kernighan, Dennis M. Ritchie: *Programmieren in C*, 1983, Hanser Verlag

[Lan64] P. J. Landin: *The Mechanical Evaluation of Expressions*, in Computer Journal **6** (4), 1964.

[Mir91] E. Miranda: *How to do machine-independent fast threaded code*, Dept. of Computer Science, Queen Mary and Westfield College, London, 1991.

[NSEP91] E. Nöcker, J. Smetsers, M. van Eekelen, M. J. Plasmeijer: *Concurrent Clean*, Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE '91), Eindhoven, The Netherlands.

[Nöc93] Eric Nöcker: *Strictness analysis using abstrakt reduction*, Technical Report, University of Nijmegen, Department of Computer Science, 1993.

[Pan] Sven Eric Panitz: *Termination Tableaux — a Calculus for Termination Proofs*, Johann Wolfgang Goethe Universität, Frankfurt/Main, 1995.

[Pey+93] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, Philip Wadler: *The Glasgow Haskell compiler: a technical overview*, Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele, 1993.

[Pey87] Simon L. Peyton Jones, ed.:*The implementation of functional programming languages*, 1987, Prentice–Hall International Series in Computer Science.

[Pey91] Simon L. Peyton Jones, John Launchbury: *Unboxed values as first class citizens*, in *Functional Programming Languages and Computer Architecture, Bosten*, Hughes ed., Lecture Notes on Computer Science 523, 1991, Springer Verlag.

[Pey92] Simon L. Peyton Jones: *Implementing lazy functional languages on stock hardware: the spineless tagless G-machine*, 1992.

[Rest] Bernd Rest: Diplomarbeit in Vorbereitung, Johann Wolfgang Goethe Universität, Frankfurt/Main, 1995.

[Ros82] J. B. Rosser: *Highlights of the history of the lambda calculus*, Proceedings of the ACM Symposium on Lisp and Functional Programming, Pittsburgh, S. 216–225, August 1982.

[SNGP91] Sjaak Smetsers, Eric Nöcker, John von Groningen, Rinus Plasmeijer: *Generating Efficient Code For Lazy Functional Languages*, University of Nijmegen, The Netherlands, May 1991.

[Sch94] Marko Schütz: *Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache*, Diplomarbeit, Johann Wolfgang Goethe Universität, Frankfurt am Main, 1994.

[Ste78] G. L. Steele: *Rabbit: a compiler for Scheme*, AI-TR-474, MIT Lab for Computer Science, 1978.

[TLA91] David Tarditi, Peter Lee, Anurag Acharya: *No assembly required: compiling Standard ML to C*, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1991.

[Wel75] P. Welch: *Some Notes on the Martin-Lof Proof of the Church Rosser Theorem as Rediscovered by Park*, Computer Lab., University of Kent, Oktober 1975.

Index