Abstrakte Maschinen zur Auswertung nicht-strikter funktionaler Sprachen: Analyse, Vergleich und Optimierung

Thorsten Buchinger

Diplomarbeit
23. August 1995

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie

Johann Wolfgang Goethe-Universität
Frankfurt am Main
Fachbereich Informatik (20)

Danksagung

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

Thorsten Buchinger

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 23. August 1995

(Thorsten Buchinger)

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ziel und Aufgabenstellung

Funktionale Programmiersprachen unterscheiden sich grundlegend von den weit verbreiteten imperativen Programmiersprachen wie Pascal oder C.

In einer funktionalen Programmiersprache zerlegen wir ein Problem nicht wie in einer imperativen Programmiersprache in eine Folge von Befehlen[1], die dieses Problem dann lösen, sondern wir erstellen eine Funktion, die das Problem löst. Diese Funktion kann Fallunterscheidungen treffen und Hilfsfunktionen aufrufen.

Für den Benutzer ist es leichter, eine Funktion in einer funktionalen Sprache zur Lösung eines Problems anzugeben als eine Reihe von Befehlen in einer imperativen Sprache, da die Funktion in der funktionalen Sprache so aufgebaut ist, wie wir die Problemlösung auch in einer natürlichen Sprache formulieren würden.

Ist eine funktionale Sprache nicht *strikt*, so haben wir die Möglichkeit mit abstrakten Elementen wie unendlichen Listen zu arbeiten. Dies ist möglich, weil nicht strikte funktionale Sprachen einen Ausdruck nur auswerten, wenn dieser gebraucht wird.

Somit können wir in einer nicht strikten funktionalen Sprache z.B. die unendliche Liste aller natürlichen Zahlen definieren und uns die ersten drei Elemente davon ausgeben lassen. In einer imperativen oder strikten Sprache würde die Definition der unendlichen Liste aller natürlichen Zahlen eine Endlosschleife zur Folge haben, da die unendliche Liste ausgewertet werden würde. In einer funktionalen Sprache erfolgt bei der Definition der unendlichen Liste keine Auswertung. Erst bei dem Befehl, die ersten drei Elemente der unendlichen Liste auszugeben, werden die ersten drei Elemente der unendlichen Liste berechnet.

[1] In einer funktionalen Sprache gibt es keine speicher- oder programmflußmodifizierenden Befehle, sondern nur Funktionsaufrufe.

Um ein Programm in einer imperativen Sprache auszuwerten, übersetzen wir die Folge von Befehlen, die das Programm darstellen, einfach in eine Folge von Maschinensprache-Befehlen und lassen diese ausführen. Dies ist kein Problem, da die Befehle in der imperativen Sprache nacheinander ausgeführt werden müssen und die Maschinensprache-Befehle vom Prozessor auch nacheinander ausgeführt werden.

Ein funktionales Programm hat aber einen anderen Aufbau: Es besteht aus einer Hauptfunktion, die ausgewertet werden soll und evtl. zur Auswertung Hilfsfunktionen aufruft. Wir können ein funktionales Programm deshalb nicht direkt in Maschinensprache übersetzen und ausführen lassen.

Um ein funktionales Programm auszuführen, brauchen wir eine *abstrakte Maschine*. Die abstrakte Maschine wertet ein funktionales Programm aus, indem sie Funktionen anwendet, Fallunterscheidungen trifft und primitive Operationen wie z.B. $a + b$ selbst berechnet.

Diese Arbeit beschäftigt sich mit dem Vergleich, der Analyse und der Untersuchung der Optimierungsmöglichkeiten von abstrakten Maschinen zur Auswertung nicht strikter funktionaler Sprachen.

1.2 Überblick

Im Vordergrund unserer Untersuchungen stehen die G-Maschine und die STG-Maschine, die zur Auswertung nicht strikter funktionaler Sprachen verwendet werden.

In **Kapitel 2** stellen wir funktionale Sprachen vor und zeigen, wie funktionale Programme durch die G-Maschine und die STG-Maschine ausgewertet werden: Die G-Maschine übersetzt jede Funktion in eine Folge von Befehlen, den G-Code, und wertet diesen dann aus. Die STG-Maschine wertet direkt das funktionale Programm aus, das vorher in eine funktionale Zwischensprache, die STG-Sprache, übersetzt wurde.

In **Kapitel 3** stellen wir einen Interpreter für die STG-Maschine vor, der funktionale Programme auswertet. Der STG-Maschinen Interpreter verschafft uns dadurch, daß er nach jedem Schritt den Zustand der abstrakten Maschine ausgibt, einen Einblick in die Arbeitsweise der STG-Maschine.

In **Kapitel 4** optimieren wir die STG-Maschine durch die Einführung von expliziten Registermanipulationsbefehlen im STG-Code.

Die Verwendung von expliziten Registermanipulationsbefehlen in STG-Programmen und die dadurch stärkere Orientierung an der Hardware ist sinnvoll, da wir bei der Ausführung in vielen Fällen unnötige Zwischenschritte sparen können.

Wir zeigen, wie wir aus jedem STG-Programm automatisch STG-Code mit abstrakten Registermanipulationsbefehlen erzeugen können.

Diesen STG-Code mit abstrakten Registermanipulationsbefehlen übersetzen wir unter Verwendung des STG-Kompilers von Frank Kiesel erst in C und dann in HP PA RISC Maschinensprache, wobei die abstrakten Registermanipulationsbefehle auf HP PA RISC Registerbefehle abgebildet werden.

In **Kapitel 5** vergleichen wir die Methoden und Konzepte der STG-Maschine mit denen der G-Maschine. Uns interessiert hierbei besonders die Frage, inwieweit sich beide Ansätze unterscheiden und inwieweit sie sich ähneln.

In diesem Zusammenhang betrachten wir zunächst die *flachen Applikationen* bei der STG-Maschine, d.h. daß Funktionen nur auf Konstanten oder Variablen angewendet werden dürfen.

Bei der G-Maschine gibt es diese Einschränkung nicht. Die G-Maschine führt aber bei der Übersetzung des funktionalen Programms in G-Code flache Applikationen ein.

Zur Untersuchung, inwieweit sich die beiden abstrakten Maschinen in der Auswertung von funktionalen Programmen ähneln, entwickeln wir einen STG-To-G Übersetzer, der es erlaubt, STG-Code in G-Code zu übersetzen, der dann auf einer (leicht modifizierten) G-Maschine ausgeführt werden kann.

Die beiden abstrakten Maschinen unterscheiden sich jedoch in Bezug auf die *Updates*. Wurde in der STG-Maschine eine Funktion ausgewertet, dann betrachtet diese zunächst das *Update-Flag*. Wenn dieses gesetzt ist, wird die Funktionsrepräsentation im Heap mit dem Ergebnis der Auswertung überschrieben. Wenn das *Update-Flag* nicht gesetzt ist, so bleibt die Funktionsrepräsentation im Heap unverändert.

Die G-Maschine führt immer ein *Update* aus, d.h. die Funktionsrepräsentation im Heap wird nach der Auswertung der Funktion immer verändert. Dies geschieht jedoch nicht durch Überschreiben mit dem Ergebnis, sondern die Funktionsrepräsentation im Heap wird mit einem *Indirektionsknoten* überschrieben, der auf das Ergebnis verweist.

Wir werden zeigen, daß wir an der G-Maschine zwei sinnvolle Optimierungen vornehmen können: Die G-Maschine kann ein funktionales Programm effizienter auswerten, wenn sie statt der *Updates mit Indirektionsknoten* die Funktionsrepräsentation im Heap direkt mit dem Ergebnis überschreibt, wie die STG-Maschine dies auch handhabt. Wir nennen diese Methode *Update in Place*.

Zum anderen können wir die G-Maschine optimieren, wenn wir sie, wie die STG-Maschine, die Updateinformation verwenden lassen: Wenn eine Funktion im Programmablauf nur einmal ausgewertet wird, so können wir bei der STG-Maschine durch ein nicht-gesetzes *Update-Flag* den Schritt, die Funktionsrepräsentation mit dem Ergebnis zu überschreiben, sparen. Der Verzicht auf dieses unnötige *Update* macht auch die G-Maschine effizienter.

Damit die G-Maschine die Updateinformation aus STG-Programmen nutzen kann, erweitern wir den STG-To-G Übersetzer.

Dieser berücksichtigt jetzt auch die *Update-Flags* in den STG-Programmen und läßt die G-Maschine *Updates in Place* ausführen, wenn das *Update-Flag* gesetzt ist und auf die Aktualisierung verzichten, wenn das *Update-Flag* nicht gesetzt ist.

Wir führen jetzt einen Vergleich der Programmauswertung der STG-Maschine und der G-Maschine, die die Updateinformation benutzt, durch und untersuchen, inwieweit sich beide Maschinen in der Programmausführung ähnlich sind.

In **Kapitel 6** stellen wir noch weitere abstrakte Maschinen vor: Die ABC-Maschine, und die *Three-Instruction Machine*.

Kapitel 2

Grundlagen

2.1 Funktionale Sprachen

Der grundlegende Unterschied zwischen funktionalen Sprachen und den (bekannteren) imperativen Programmiersprachen, wie Pascal oder C, ist, daß funktionale Sprachen keine Befehle kennen, sondern nur Funktionsaufrufe.

Ein Programm in einer imperativen Programmiersprache, das die Zahlen von 1 bis 10 aufaddiert, sieht so aus:

```
summe := 0
for i := 1 to 10 do
summe := summe + i
```

In einer funktionalen Sprache geben wir die Funktion *summe* an, die wir mit *summe(10)* aufrufen:

```
summe(0) = 0
summe(n) = n + summe(n-1)
```

Auch gibt es in funktionalen Programmen keine globalen Variablen, die wir mit Werten belegen könnten. Ein funktionales Programm besteht aus Funktionen, die sich gegenseitig aufrufen können; Werte werden dabei nicht in globalen Variablen gespeichert, sondern immer nur als Parameter übergeben.

Wir müssen alle Werte, die im weiteren Ablauf des Programms noch gebraucht werden, durch Übergabe als Parameter bei den Funktionsaufrufen angeben, da diese sonst verloren gehen. Variablen sind lediglich als lokale Variablen in Funktionen erlaubt, um Zwischenergebnisse zum leichteren Zugriff mit Namen zu versehen. Diese lokalen Variablen dürfen nicht verändert werden.

Zur Zeit aktuelle funktionale Sprachen sind Miranda [Tur85], [Hol91] und [Hin92], LML [Aug84], Haskell [Dav92] und [PJHH+92], Gofer [Jon91],[Cun94] und [Jon94] und Clean [NSvP91].

2.1.1 Haskell und Gofer

Haskell wurde im Haskell-Report [HPW+92] definiert. Gofer wurde von Mark P. Jones [Jon94] entwickelt und ist syntaktisch und semantisch an Haskell angelehnt, verzichtet aber auf einige (nicht unbedingt notwendige) Leistungsmerkmale von Haskell.

Dadurch entstand ein sehr kompakter Interpreter/Kompiler, der durch seinen geringen Ressourcenbedarf auch auf PCs lauffähig ist.

Leistungsmerkmale, die sowohl Haskell als auch Gofer aufweisen, sind:

- Die Sprachen sind rein funktional.

  Dies bezieht sich insbesondere darauf, daß keine Befehle wie in imperativen Sprachen, sondern nur Funktionsaufrufe und Funktionsdefinitionen erlaubt sind.

  Auch für Gofer und Haskell trifft das für rein funktionale Sprachen kennzeichnende Merkmal zu, daß diese Sprachen keine globalen Variablen zulassen. Die Sprachen *Scheme* [AS85], *LISP* [Ste90] und *ML* [Aug84] sind z.B. nicht rein funktional, da es in diesen Sprachen Befehle gibt, um globale Variablen festzulegen und zu verändern[1].

- Die Sprachen sind stark getypt.

  Beide Sprachen haben ein Typsystem, das für jede Funktion angibt, welchen Typ ihre Argumente haben und welchen Typ die Funktion zurückgibt.

  Die schon angegebene Funktion *summe* hat den Typ $summe :: Int \rightarrow Int$, denn sie erhält eine Integerzahl als Argument und gibt eine Integerzahl zurück. Erhält eine Funktion zwei Argumente, z.B. eine Additionsfunktion: $add\, x\, y = x + y$ so ist ihr Typ: $add :: Int \rightarrow Int \rightarrow Int$.

  Diese Typendefinition wird verständlich, wenn wir in Betracht ziehen, daß funktionale Sprachen *currying* verwenden, d.h. die Funktion *add*, die zwei Argumente erwartet und diese addiert, nimmt das erste Integerargument und verwandelt sich dann in eine neue Funktion, die nur noch ein Argument erwartet und darauf das vorher schon genommene Integerargument addiert.

[1] Z.B. können in LISP mit *setq* Variablenzuweisungen getätigt werden.

- Die Sprachen sind nicht strikt.

  Eine nicht strikte Sprache, die mit *sharing*[2] und *update*[3] arbeitet, nennen wir auch *lazy*. Das Kennzeichen einer nicht strikten Sprache ist es, daß sie nur Ausdrücke auswertet, die auch benötigt werden.

  Wenn wir z.B. in einer nicht strikten Sprache den Ausdruck *first*[4] *(10,1/0)* auswerten lassen, dann wird der Ausdruck *1/0* niemals berechnet und somit gibt es auch keinen Fehler.

  Eine wichtige Methode der *lazy* Auswertung von Programmen ist die Auswertung zur *Weak Head Normal Form*. Die *Weak Head Normal Form* (WHNF) wird in [PJ87] wie folgt definiert:

  Definition:
  Eine Lambdaform ist genau dann in *Weak Head Normal Form*, wenn sie die Form:
  F E_1 E_2 ... E_n
  hat, wobei n$\geq$0, und F entweder eine Variable, ein Datenobjekt, eine Lambdaform oder eine eingebaute Funktion ist und
  F E_1 E_2 ... E_m
  für kein m$\leq$n ausgewertet werden kann[5].

  Ausdrücke in *WHNF* sind z.B.: Die Zahl 3 oder eine *Cons* Zelle, weil diese Ausdrücke nicht weiter ausgewertet werden können, der Ausdruck $(3-4)+$, weil das + nicht genug Argumente hat und der Ausdruck $(\lambda x.5 + 1)$, weil die Lambdaform auf nichts angewendet wird.

  Die beiden letzten Ausdrücke haben innere Ausdrücke, die reduziert werden können. Ist ein Ausdruck nicht weiter reduzierbar und hat er keine inneren Ausdrücke, die reduziert werden können, so ist der Ausdruck in *Normal Form*.

  Die beiden letzten Ausdrücke sind in WHNF aber nicht in *Normal Form*, weil diese innere Ausdrücke enthalten, die reduziert werden können.

  In lazy-funktionalen Sprachen wollen wir nur Ausdrücke auswerten, die auch gebraucht werden. Dies erreichen wir dadurch, daß wir die Ausdrücke nur zur WHNF auswerten.

  Marko Schütz beschäftigte sich in seiner Diplomarbeit[Sch94] damit, wie man effizient herausfinden kann, welche Ausdrücke ausgewertet werden müssen und welche nicht.

[2] Wenn zwei Funktionen das gleiche Argument haben, dann wird dieses nicht kopiert, sondern beide Funktionen haben einen Zeiger auf das gleiche Argument.

[3] Wenn das Argument ein Ausdruck ist, der ausgewertet wurde, dann wird seine Repräsentation durch das Ergebnis ersetzt.

[4] Die Funktion *first* liefert das erste Argument eines Paares.

[5] Ein Ausdruck kann nicht ausgewertet werden, wenn nicht genug Argumente vorhanden sind oder der Ausdruck bereits ausgewertet ist.

- Die Sprachen erlauben Funktionen höherer Ordnung.

  Funktionen höherer Ordnung sind Funktionen, die andere Funktionen als Argumente nehmen oder andere Funktionen als Ergebnis liefern. Ein Beispiel hierfür ist die *nach* Funktion, die zwei Funktionen als Argumente nimmt und eine Funktion als Ergebnis liefert, die aus der Hintereinanderausführung der beiden Funktionen besteht. Der Typ der *nach* Funktion ist:

  $nach :: (b \to c) \to (a \to b) \to (a \to c).$

  Dabei ist $(b \to c)$ der Typ der ersten Funktion und $(a \to b)$ der Typ der zweiten Funktion. a,b und c sind Typvariablen, die für beliebige Typen stehen.

  Der Typ der *nach* Funktion legt fest, daß der Ergebnistyp der zweiten Funktion gleich dem Typ des Arguments der ersten Funktion entsprechen muß und daß die Ergebnisfunktion ein Argument vom Typ des Argumentes der zweiten Funktion nimmt und ein Ergebnis vom Typ der ersten Funktion liefert.

- Die Sprachen erlauben Pattern Matching.

  Eine Funktion der Form:

  $f\ pat_1 ... pat_n = expr$

  benutzt *Pattern Matching*. D.h. die Argumente, mit denen die Funktion f aufgerufen wird, werden mit den Pattern $pat_1 ... pat_n$ verglichen und f liefert das Ergebnis *expr*, falls die Argumente mit den Pattern übereinstimmen.

  Wir können damit die Funktion f in mehreren Gleichungen definieren, so daß sie, je nachdem mit welchen Argumenten sie aufgerufen wird, ein anderes Ergebnis liefert.

  Die Gofer Funktion *not* wurde z.B. mit Pattern Matching geschrieben:

  not True = False; not False = True.

- Die Sprachen erlauben benutzerdefinierte Datentypen.

  Datentypen können in Gofer selbst definiert werden. Diese können wir dann genauso wie eingebaute Datentypen verwenden. Es ist auch möglich, Pattern Matching für benutzerdefinierte Datentypen auszuführen.

- Die Sprachen erlauben überladene Funktionen.

  Eine *überladene Funktion* ist z.B. die = Funktion, die z.B. sowohl für Integerzahlen, als auch für Zeichenfolgen die Gleichheit liefert. Haben wir einen benutzerdefinierten Datentypen verwendet, so können wir die Gofer-Gleichheitsfunktion durch Hinzufügen einer neuen *Instanz* auch für unseren selbstdefinierten Datentyp verfügbar machen.

- Die Sprachen erlauben *List Comprehensions*.

  Mit *List Comprehensions* können wir auf einfache Weise eine Liste erstellen lassen. Wir schreiben beispielsweise einfach:

  $[(x, y)|x \leftarrow [1..3], y \leftarrow [1..2]]$,

  um die zweidimensionale Liste der Form *(x,y)*, wobei x von 1 bis 3 und y von 1 bis 2 läuft, zu erhalten. Der Vorteil von *List Comprehensions* liegt in der sehr kompakten Notation.

- Die Sprachen erlauben Benutzerdialoge.

  Die einfachste Art, Ein- und Ausgabe in funktionalen Sprachen zu realisieren, ist unproblematisch: Da das Programm durch einen Funktionsaufruf gestartet wird, werden die Eingabedaten der Funktion als Argumente mitgegeben. Die Ausgabe stellt ebenfalls kein Problem da.

  Soll jedoch in einer funktionalen Sprache ein Benutzerdialog geführt werden, d.h. dem Benutzer die Möglichkeit eingeräumt werden, während des Programmlaufs Daten einzugeben, so ist dies mit einigen Schwierigkeiten behaftet: Dies liegt an der sogenannten *referentiellen Transparenz*, die in jedem funktionalen Programm aufrechterhalten werden muß.

  Referentielle Transparenz bedeutet, daß eine Funktion, egal in welchem Kontext sie mit einer bestimmten Eingabe aufgerufen wird, immer das gleiche Ergebnis liefern muß. Wenn wir eine Gofer Funktion einfach eine Eingabe lesen lassen würden, so würde diese keinesfalls immer die gleiche Ausgabe liefern und die referentielle Transparenz wäre zerstört.

  Die Gofer I/O Funktionen gewährleisten, daß die referentielle Transparenz erhalten bleibt.

  Dazu geht Gofer den Umweg, daß eine Gofer Funktion, die eine Eingabe liest, vom Typ *Dialogue* sein muß. Der Typ von *Dialogue* ist:

  $Dialogue :: [Response]- > [Request]$

  Eine Funktion vom Typ *Dialogue* nimmt eine Liste von Antworten und erzeugt eine Liste von Anfragen. Eine Antwort ist dann z.B. eine Zeichenkette, die die Eingabe darstellt und eine Anfrage z.B. der Befehl, von der Tastatur etwas zu lesen. Diese Vorgehensweise, die Antwort zu nehmen und darauf die Frage zu stellen, ist zwar komplizierter als in imperativen Programmiersprachen, dafür bleibt aber die referentielle Transparenz erhalten.

2.2 Kompilierung von funktionalen Sprachen

Wir wollen ein Programm in einer stark getypten, rein funktionalen nicht strikten Sprache wie Haskell oder Gofer kompilieren. Dazu durchlaufen wir die folgenden drei Schritte, wobei wir uns jeden Schritt auch als Abstraktionsebene vorstellen können:

- Als oberste Abstraktionsebene steht das Quellprogramm in der starkgetypten, rein funktionalen nicht strikten Sprache wie Haskell oder Gofer.
- Die zweite Abstraktionsebene bildet die Core-Language, in die das Programm jetzt übersetzt wird. Die Core-Language bildet dabei den Sprachkern einer funktionalen Sprache, oder auch eine minimale funktionale Sprache, denn der syntaktische Zucker von Haskell ist nun auf elementare Ausdrücke reduziert, die Funktionen wurden auf die Richtigkeit ihrer Typen überprüft und Überladungen wurden aufgelöst. Außerdem wurde das Pattern-Matching in *case*-Anweisungen transformiert.

  Wenn das Core-Language Programm später durch die STG-Maschine ausgeführt werden soll, dann wird die Core-Language jetzt in die STG-Sprache übersetzt. Da hier keine entscheidenden Änderungen vollzogen werden, können wir auch die STG-Sprache noch zur zweiten Abstraktionsebene zählen.

  Soll später unser Core-Programm durch die G-Maschine ausgeführt werden, dann erfolgt keine weitere Übersetzung der Core-Language, weil die G-Maschine ein Core-Language Programm als Eingabe nimmt, in G-Code übersetzt und dann ausführt.
- In der dritten Abstraktionsebene wird das transformierte funktionale Programm ausgeführt. Die Anweisungen, wie das transformierte Programm auszuführen ist, werden durch eine sog. *abstrakte Maschine*, die STG-Maschine oder die G-Maschine gegeben.

  Die Anweisungen der abstraken Maschinen können mit dem funktionalen Programm zu einem ausführbaren Maschinensprache- oder C-Programm übersetzt werden. Die Übersetzung nach C wird vor allem aus Portabilitätsgründen gewählt[6].

Wir können für die abstrakten Maschinen aber auch Interpreter erstellen. Der STG-Maschinen Interpreter führt den STG-Code direkt aus. Der G-Maschinen Interpreter verlangt, daß die Core-Language vor der Ausführung in G-Code übersetzt wird (siehe Abbildung 2.1).

In dieser Arbeit, die sich mit dem Vergleich, der Analyse und der Optimierung von abstrakten Maschinen, die zur Auswertung funktionaler Sprachen gebraucht werden, beschäftigt, stehen zwei abstrakte Maschinen im Vordergrund: die STG-Maschine und die G-Maschine.

[6] In der Praxis kommt es dennoch zu Portierungsproblemen, wie z.B. beim Glasgow Haskell Kompiler.

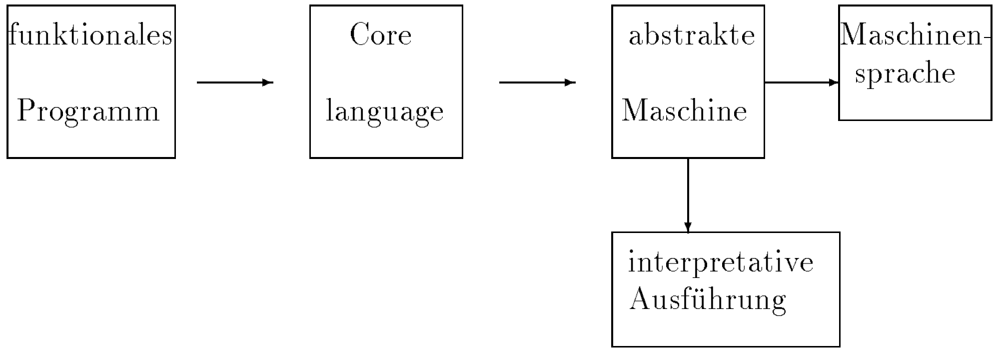

Abbildung 2.1: Kompilierung von funktionalen Sprachen

Um die STG-Maschine zu verstehen, ist die Kenntnis der STG-Sprache erforderlich. Wir stellen im nächsten Abschnitt zuerst die G-Maschine und dann die STG-Sprache und die STG-Maschine vor.

2.3 Die G-Maschine

Die G-Maschine wurde am Chalmers Institute of Technology, Göteborg, Schweden von Augustsson und Johnsson entwickelt.

In dem Buch [PJL91] von Simon L Peyton Johnes und David R Lester wird die G-Maschine vorgestellt und gezeigt, wie man die abstrakte G-Maschine implementiert, so daß man einen Interpreter für funktionale Programme hat.

Ein funktionales Programm, bestehend aus Funktionen und Argumenten, wie z.B. **main = head(Cons(1,Nil))**, kann auch als Graph dargestellt werden. Die Auswertung eines als Graph dargestellten funktionalen Programms durch Anwendung der Funktionen auf ihre Argumente nennen wir *Graph Rewriting* (siehe Abbildung 2.2).

Die grundlegende Idee der G-Maschine ist es, jede Funktionsdefinition in eine Sequenz von Instruktionen zu übersetzen, die die *Graph Rewriting* Operationen der Funktion durchführen.

Um ein funktionales Programm mit der G-Maschine auszuführen, durchlaufen wir zwei Schritte: Zuerst wird jede Funktion des Programms in eine Folge von Instruktionen übersetzt, den *G-Code*, dann wird das Programm durch Abarbeitung dieser G-Befehle ausgewertet.

Das ursprüngliche funktionale Programm, das der G-Maschine in der Zwischensprache Core-Language vorliegt, wird zur Ausführung nicht mehr benötigt.

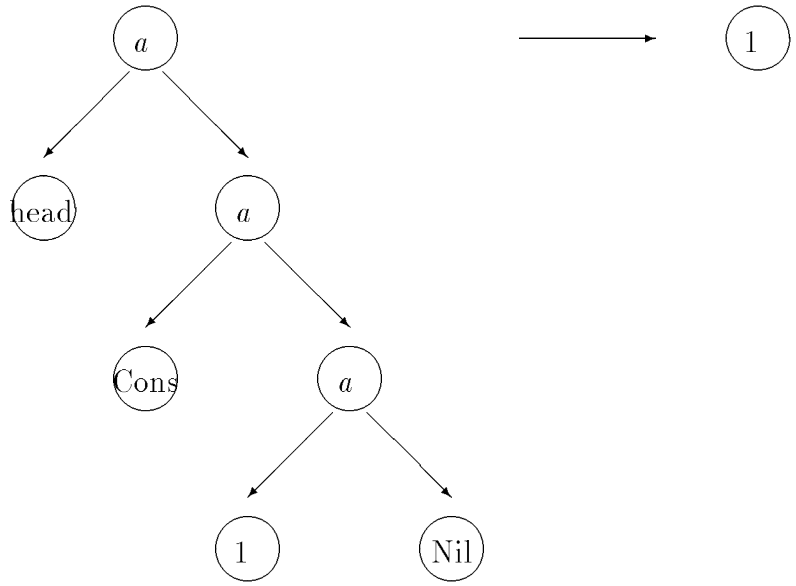

Abbildung 2.2: Darstellung und Auswertung eines funktionalen Programms als Graph. *Applikations-Knoten* sind durch *a* gekennzeichnet.

2.3.1 Die Komponenten der G-Maschine

Die G-Maschine besteht aus dem auszuführenden *G-Code*, einem *Stack*, dem *Dump*, einem *Heap*, den *Globals* sowie einer Statistikkomponente.

- **Der G-Code:**

  Jede Funktion des funktionalen Programms, das mit der G-Maschine ausgewertet werden soll, wird zu einer Folge von *G-Code* Befehlen übersetzt.

  Diese Befehle werden dann nacheinander abgearbeitet, so daß dadurch die Funktion ausgeführt wird. Die genaue Beschreibung aller G-Befehle befindet sich im nächsten Abschnitt.

- **Der Dump:**

  Der *Dump* dient dazu, die *Stack*- und die *Code*-Komponenten der G-Maschine zwischenzuspeichern, um einen Ausdruck evaluieren zu können. Ist der Ausdruck evaluiert, werden die *Stack*- und *Code*-Komponenten wiederhergestellt.

- **Die Globals-Komponente:**

  Jede Funktion in der Core-Language wird in eine Sequenz von G-Maschinen Befehlen übersetzt. Eine Funktion nennen wir hier *Superkombinator*. Diese

Befehlssequenz wird im Heap an einer bestimmten Adresse gespeichert. Um zu wissen, wo die Befehlssequenz eines Superkombinators steht, bildet die *Globals*-Komponente die Namen der Superkombinatoren auf Heap-Adressen ab.

- **Der Heap:**

  Der *Heap* besteht aus Heap-Knoten, die folgende Informationen enthalten können:

  Integerzahlen, *Applikationen*, *G-Code Sequenzen*, die z.B. zu einem Superkombinator gehören, *Indirektionsknoten*, die auf einen anderen Heap-Knoten verweisen und *Konstruktoren*.

- **Der Stack:**

  Der *Stack* arbeitet wie in einer bekannten Stack-Maschine: Er dient dazu, den Code der Superkombinatoren auszuführen. Benötigt ein Superkombinator Argumente, so müssen diese vor der Ausführung des G-Codes für diesen Superkombinator auf dem *Stack* liegen. Der Superkombinator verändert den *Stack* und nach der Ausführung des G-Codes für den Superkombinator steht das Ergebnis auf dem *Stack*.

2.3.2 Der G-Code

Der G-Code kennt folgende Befehle:

- *Push num*

  Die *Push* Anweisung ermöglicht es, auf Stackelemente zuzugreifen. *Push* n holt ein Argument aus dem Stack und schreibt es an die oberste Stelle. Dabei wird das Element, das vor dem Aufruf das oberste Element ist, nicht mitgezählt, das zweite hat die Nummer 0 usw.

- *Pop num*

  Pop n entfernt die obersten n Stackelemente.

- *Mkap*

  Die *Mkap* Anweisung nimmt die beiden obersten Stack-Einträge aus dem Stack und schreibt an die oberste Stelle einen neuen Stack-Eintrag, der auf einen Heap-Knoten zeigt, welcher die Applikation des obersten Stack-Eintrages auf den zweitobersten ist.

- *Eval*

Die *Eval* Anweisung evaluiert das oberste Stackelement zur *Weak Head Normal Form.* Der Stack und die Code-Komponente der G-Maschine werden vorher auf dem Dump zwischengespeichert.

- *Pushglobal name*

  Die *Pushglobal name* Anweisung schreibt an die oberste Stelle im Stack einen Zeiger auf den G-Code für den Superkombinator *name*.

- *Pushint num*

  Die Anweisung *Pushint num* erzeugt einen Heap-Knoten, der die Zahl *num* enthält und setzt an die oberste Stelle des Stacks einen Zeiger, der auf diesen Heap-Knoten zeigt.

- *Update num*

  Update num überschreibt den $num + 1$. Stack-Eintrag mit einem Indirektionsknoten, der auf den obersten Stack-Eintrag zeigt.

  Diese Anweisung führt ein *Update durch Indirektionsknoten* aus: An der obersten Stackposition steht das Ergebnis mit dem der $n + 1$. Stack-Eintrag (die Funktionsdefinition) aktualisiert werden soll.

- *Unwind*

  Die *Unwind* Anweisung befiehlt der G-Maschine den obersten Stack-Eintrag weiter zu evaluieren. Sie hat ihren Namen von der Betrachtung des funktionalen Programms als Graph: Mit *Unwind* wird dieser Graph *abgerollt*.

  Ist der oberste Knoten im Graph ein Applikationsknoten, so wird der links von diesem Applikationsknoten stehende Knoten zur Auswertung gebracht.

  Ein Funktionsname an oberster Stelle wird durch den Graph der Funktion ersetzt.

- *Slide num*

  Wenn die G-Maschine eine Funktion ausführt, schreibt sie zuerst die Argumente, die die Funktion benötigt, auf den Stack und bildet dann auf dem Stack das Ergebnis.

  Mit der Anweisung *Slide* n wird der oberste Stack-Eintrag beibehalten, aber die n darunterliegenden Stack-Einträge werden gelöscht.

  Sinnvoll ist die Anweisung, wenn auf dem obersten Stack-Eintrag bereits das Ergebnis erzeugt wurde und in den n darunterliegenden Stack-Einträgen die Argumente stehen, die wir nun nicht mehr brauchen.

- *Alloc num*

  Alloc n erzeugt n leere Heap-Einträge und schreibt die Adressen oben auf den Stack. Die leeren Heap-Einträge werden für lokale Variablen verwendet.

- *Pack tag arity*

  Pack tag arity erzeugt einen neuen Konstruktor-Heap-Knoten wobei der *tag* des Konstruktors diesen identifiziert und die *arity* des Konstruktors angibt, wieviele Argumente der Konstruktor bindet.

  Die Anweisung *Pack tag arity* nimmt die obersten *arity* Elemente vom Stack und schreibt diese als Konstruktor-Argumente in den neuen Heap-Knoten.

- *Casejump [(num,gmCode)]*

  Casejump erwartet, daß das oberste Stackelement ein Konstruktor ist. Dann wird aus der Liste mit *(num1,gmCode),(num2,gmCode),...* die Integerzahl *num* gesucht, die dem *Tag* des Konstruktors entspricht und der entsprechende G-Code wird ausgeführt.

- *Split num*

  Die *Split* Anweisung wird benötigt, um auf die Argumente eines Konstruktors zugreifen zu können. *Split num* schreibt die ersten *num* Argumente des Konstruktors, der vor dem Aufruf von *Split* an der obersten Stackposition stehen muß, oben auf den Stack.

- *Print*

  Die *Print* Anweisung gibt das oberste Stackelement aus. Dies ist im einfachsten Fall eine Integerzahl. Ist das oberste Stackelement ein Konstruktor, so wird die Anweisung gegeben, jedes der Konstruktor-Argumente nacheinander auszuwerten und auszugeben.

- *Add, Sub, Mul, Div*

  Die Rechenoperationen nehmen die beiden obersten Stackelemente, wenden die entsprechende Operation an und schreiben an die oberste Stackposition einen Zeiger auf einen neuen Heap-Knoten, der das Ergebnis enthält.

- *Neg*

  Die*Neg* Anweisung negiert das oberste Stackelement.

- *Eq, Ne, Le, Lt, Ge, Gt*

  Die Vergleichsfunktionen nehmen die beiden obersten Stackelemente und liefern als neues oberstes Stackelement die Zahl 1, wenn der Vergleich positiv ausfiel und die Zahl 0, wenn der Vergleich negativ ausfiel.

- *Cond gmCode gmCode*

  Die *Cond* Anweisung dient zur Verzweigung, nachdem ein Vergleich ausgeführt wurde. Sie enthält zwei Code-Sequenzen. Steht auf dem Stack an erster Stelle die Zahl 1, so wird die erste Code-Sequenz ausgeführt. Steht dort die Zahl 0, so wird die zweite Code-Sequenz ausgeführt.

2.4 Die STG-Sprache

Die STG-Sprache ist eine primitive funktionale Sprache. Es ist zwar möglich, Programme direkt in der STG-Sprache zu schreiben, doch aufgrund ihrer fehlenden Leistungsmerkmale, wie z.B. Pattern-Matching und der schwierigen Notation für die Variablenübergabe schreiben wir die Programme in einer funktionalen Sprache wie Gofer oder Haskell und benutzen die STG-Sprache als Zwischensprache.

Wie Programme in der STG-Sprache interpretativ auszuführen sind, beschreibt die operationale Semantik. Die operationale Semantik wird in Form von Zustandsübergängen angegeben und auch abstrakte Maschine genannt. Die abstrakte Maschine für die STG-Sprache nennen wir STG-Maschine.

Der Name STG-Sprache bzw. STG-Maschine resultiert daraus, daß die STG-Maschine eine Weiterentwicklung der G-Maschine [PJL91] ist. Die G-Maschine muß den auszuführenden Code zuvor auf einen sog. *Spine* kopieren und unterscheidet Heap-Objekte durch sog. *Tags*.

Da die STG-Maschine keinen *Spine* benötigt, weil sie mit einem Zeiger direkt zu dem auszuführenden Code springen kann und die Heap-Objekte alle die gleiche Repräsentation haben, also nicht durch *Tags* unterschieden werden, heißt sie *Spineless-Tagless G-Machine*.

2.4.1 Syntax der STG-Sprache

Die Syntax der STG-Sprache, die wir im Rahmen dieser Arbeit verwenden, zeigt Abbildung 2.3.

Sie basiert auf der von Simon L. Peyton Jones in [PJ92] angegebenen Syntax für die STG-Sprache, an der wir folgende Änderungen vorgenommen haben:

- Die Argumente einer STG-Funktion werden nicht auf einem *Argument-Stack* gespeichert, sondern auf zwei Stacks aufgeteilt: Ist das Argument ein Integerwert (*unboxed value*), dann wird es auf dem *B-Stack* gespeichert. Ist das Argument ein Adresswert (*boxed value*), dann wird es auf dem *A-Stack* gepeichert.

  Wir haben deshalb in der Syntax die Darstellung der Argumente einer STG-Funktion um die Angabe, auf welchem Stack jedes Argument steht, erweitert.

- Wir haben die Syntax der *case*-Anweisung geändert: Es war in der in [PJ92] angegebenen Syntax zwingend notwendig, daß eine *Default-Alternative* angegeben wurde, auch wenn diese gar nicht benutzt wurde. Die *Default-Alternative* darf jetzt weggelassen werden.

- Die verschiedenen Alternativen einer *case*-Anweisung werden nicht mehr durch Semikolons getrennt, sondern durch Leerzeichen. Um die *case*-Anweisungen noch eindeutig parsen zu können, muß hinter der letzten Alternative das Schlüsselwort *endcase* stehen. Dies hat den Nebeneffekt, daß nicht mehrere Semikolons hintereinander in STG-Programmen auftauchen, die verwirrend und unschön sind.

- Algebraische Alternativen (Konstruktoren) müssen in der *case*-Anweisung mit dem Schlüsselwort *Algebraic* gekennzeichnet werden, primitive Alternativen (Integerwerte) mit dem Schlüsselwort *Primitive*. Dies ist notwendig, um die Alternativen bei der Implementierung eindeutig parsen zu können.

Ein STG-Programm wird durch eine beliebige Anzahl von *Bindings* dargestellt, wobei ein *Binding* eine Funktionsbindung ist. Bei einer solchen Funktionsbindung wird ein Variablenname an eine *Lambdaform* gebunden.

Eine Lambdaform wird als: $vars\ \pi\ args \rightarrow expr$ dargestellt. Dabei ist *vars* die Menge der freien Variablen, π ist das Update-Flag, *args* ist die Menge der Argumente und *expr* ist ein STG-Ausdruck, wie er in Abbildung 2.3 erklärt wird.

Anhand der Existenz von freien Variablen können wir erkennen, daß die Umwandlung von Lambdaformen (mit freien Variablen) in Funktionen (ohne freie Variablen), das sog. *Lambda-lifting*, in der STG-Sprache nicht notwendig ist. D.h. in der STG-Sprache können Funktionen freie Variablen haben.

Das *Update-Flag* einer Lambdaform gibt an, ob die Funktionsrepräsentation, die im Heap abgelegt wird, nach ihrer Auswertung mit dem Ergebnis überschrieben werden soll.

Außer der speziellen Form der Funktionsbindung, gibt es noch andere Charakteristiken der STG-Sprache:

- Alle Funktions- und Konstruktor-Argumente sind einfache Variablen oder Konstanten. Wenn wir eine Funktion auf ein nicht-triviales Argument anwenden wollen, müssen wir in der Funktionsanwendung eine einfache Variable verwenden, der wir vorher mit der *let* Anweisung den (nicht-trivialen) Ausdruck zugewiesen haben.

- Alle Konstruktoren und eingebauten Operationen sind *gesättigt*. Ein Konstruktor bzw. eine eingebaute Operation ist gesättigt, wenn alle benötigten Argumente zur Verfügung stehen. Dadurch wird die operationale Semantik der STG-Sprache leichter.

  Wir können ungesättigte Funktionen und Konstruktoren leicht in gesättigte transformieren, wenn wir weitere Lambdaformen um den Konstruktor oder die Funktion herum bilden.

<table>
<tr><td>STG-Programm:</td><td>prog</td><td>→ binds</td></tr>
<tr><td>Bindings:</td><td>binds</td><td>→ var "=" lf {var "=" lf}</td></tr>
<tr><td>Lambdaform:</td><td>lf</td><td>→ vars "\"π args "→" expr</td></tr>
<tr><td>Update-Flag:</td><td>π</td><td>→ "u"
| "n"</td></tr>
<tr><td>Argumente:</td><td>args</td><td>→ "{" stack var { "," stack var } "}"</td></tr>
<tr><td>Stacks:</td><td>stack</td><td>→ "A" | "B"</td></tr>
<tr><td>STG-Ausdruck:</td><td>expr</td><td>→ "let" binds "in" expr
| "letrec" binds "in" expr
| "case" expr "of" alts "endcase"
| var atoms
| constr atoms
| prim atoms
| literal</td></tr>
<tr><td>Alternativen:</td><td>alts</td><td>→ "Algebraic" aalts
| "Primitive" palts</td></tr>
<tr><td>Algebraische Alternativen:</td><td>aalts</td><td>→ constr vars "→" expr | default
{ constr vars "→" expr | default }</td></tr>
<tr><td>Primitive Alternativen:</td><td>palts</td><td>→ literal "→" expr | default
{ literal "→" expr | default }</td></tr>
<tr><td>Default-Alternative:</td><td>default</td><td>→ var "→" expr
| "default" "→" expr</td></tr>
<tr><td>Literale:</td><td>literal</td><td>→ *Ziffer* { *Ziffer* }</td></tr>
<tr><td>Integeroperationen:</td><td>prim</td><td>→ "+" | "−" | "*" | "/"
| "%" | "=" | "<" | "<="
| ">" | ">=" | "! =" | "sqrt"</td></tr>
<tr><td>Variablenliste:</td><td>vars</td><td>→ var {"," var}</td></tr>
<tr><td>Listen von Atomen:</td><td>atoms</td><td>→ atom {"," atom}</td></tr>
<tr><td>Atome:</td><td>atom</td><td>→ var | literal</td></tr>
<tr><td>Variablen:</td><td>var</td><td>→ *Buchstabe* { *Buchstabe* }</td></tr>
<tr><td>Konstruktoren:</td><td>constr</td><td>→ "<"*Ziffer*">"</td></tr>
</table>

Abbildung 2.3: Die Syntax der STG-Sprache

```
main = {} \u {} -> zweimal 2
zweimal = {} \n {B x} -> * {2,x}
```

Abbildung 2.4: Ein einfaches STG-Programm

- Es gibt kein *Pattern-Matching*, sondern nur *case*-Anweisungen. Mit *Pattern-Matching* realisierte Fallunterscheidungen werden in der STG-Sprache durch *case*-Anweisungen erreicht.
- Die STG-Sprache unterstützt *unboxed values*. Wenn wir mit einem Wert, der als *unboxed value* dargestellt ist, arbeiten, dann arbeiten wir nicht mit der Adresse des Wertes (bei der Übergabe und bei Rechnungen), sondern mit dem Wert selbst.

  Wenn wir mit der Adresse eines Wertes arbeiten, so nennen wir dies im Unterschied dazu *boxed value*.

Unter den *Bindings*, die das STG-Programm bilden, gibt es immer eine Funktionsbindung, in der der Variablenname *main* an eine Lambdaform gebunden wird. Die Variable *main* und die anderen Variablen, die durch eine Funktionsbindung an eine Lambdaform gebunden werden, nennen wir *globale Variablen*.

Werden in einem STG-Programm an anderer Stelle auch Variablen gebunden, so nennen wir diese *lokale Variablen*. Der Wert der globalen Variablen *main* ist der Wert des STG-Programms.

Als Beispiel betrachten wir ein einfaches STG-Programm (siehe Abbildung 2.4).

In dem Programm wird die Funktion *zweimal* mit dem Integerwert 2 als Argument aufgerufen. Die Funktion *zweimal* nimmt ein Argument und liefert das Argument multipliziert mit 2 als Ergebnis. Das Programm wird in der STG-Syntax gespeichert, indem die beiden globalen Variablen *main* und *zweimal* definiert werden.

An der Definition der globalen Variablen *main* in STG-Syntax erkennen wir, daß *main* keine freien Variablen hat (erster leerer Klammerausdruck), die Funktionsrepräsentation im Heap nach der Auswertung mit dem Ergebnis überschrieben werden soll (angegeben durch $\backslash u$) und daß *main* keine Argumente nimmt (zweiter leerer Klammerausdruck).

Der Ausdruck, der unter der globalen Variablen *main* gespeichert wird, besteht aus der Funktionsanwendung der Funktion *zweimal* auf den Integerwert 2.

Die Funktion *zweimal*, die unter der globalen Variablen *zweimal* in STG-Syntax gespeichert wird, hat keine freien Variablen (erster leerer Klammerausdruck), ihre Repräsentation im Heap soll nach der Auswertung nicht mit dem Ergebnis überschrieben werden ($\backslash n$) und die Funktion nimmt ein Argument vom B-Stack, das sie unter der Bezeichnung x zur Verfügung stellt (angegeben durch: {B x}).

Der STG-Ausdruck, der zu der Funktion *zweimal* gehört, ist $*\{2, x\}$. Der Stern ($*$) steht dabei für die eingebaute Multiplikationsfunktion, die die beiden Argumente multipliziert. Die Funktion heißt eingebaut, weil die abstrakte Maschine die Multiplikation selbständig ausführt.

2.5 Die abstrakte STG-Maschine

Um die abstrakte STG-Maschine vorzustellen, beantworten wir drei Fragen:
In welcher Form liegt ein STG-Programm der STG-Maschine vor ?
Aus welchen Komponenten setzt sich die STG-Maschine zusammen ?
Wie wird die operationale Semantik auf ein STG-Programm angewendet ?

2.5.1 Repräsentation von STG-Programmen

Ein STG-Programm ist eine Menge von globalen Variablendefinitionen. Dabei wird jeder globalen Variablen eine Lambdaform, bestehend aus freien Variablen, Update-Flag, Argumenten und einem Ausdruck zugewiesen. Das in Abbildung 2.4 vorgestellte STG-Programm wird damit zu der Zuordung:

globale Variable	Lambdaform
main	{} $\backslash u$ {} $\rightarrow$ zweimal{2}
zweimal	{} $\backslash n$ {B x} $\rightarrow$ *{2,x}

2.5.2 Komponenten der STG-Maschine

Die STG-Maschine besteht aus folgenden Komponenten:

- *Der Code.* Der Code ist nicht das STG-Programm, sondern besteht aus Anweisungen, wie die STG-Maschine das STG-Programm auszuführen hat.
- *Der A-Stack.* Der A-Stack speichert Adressen, die Argumente für Funktionen sind.
- *Der B-Stack.* Der B-Stack speichert Integerwerte, die Argumente für Funktionen sind.
- *Der Return-Stack.* Der Return-Stack speichert weiterführenden Code zu dem nach einer *case*-Anweisung verzweigt wird.

- *Der Update-Stack.* Der Update-Stack speichert Update-Informationen.
- *Der Heap.* In Heap-Knoten werden STG-Ausdrücke gespeichert.
- *Die globale Umgebung.* Die globale Umgebung bildet globale Variablen auf Heap-Knoten ab.

Der Code

Der Code besteht aus folgenden Befehlen:

- Eval e ρ. Der *Eval* Befehl wertet den Ausdruck e in der lokalen Umgebung ρ aus.

  Die lokale Umgebung ist dabei eine Abbildung von Variablennamen auf Werte. Die Umgebung ist lokal, weil sie, im Gegensatz zur immer vorhandenen globalen Umgebung, nur temporär existent ist.

- Enter a. Der *Enter* Befehl wendet den Heap-Knoten mit der Adresse a auf die Argumente auf dem Argument-Stack an.

- ReturnCon c ws. Der *ReturnCon* Befehl gibt den Konstruktor c, angewendet auf die Werte ws zu der weiterführenden Code-Sequenz auf dem Return-Stack zurück.

- ReturnInt k. Der *ReturnInt* Befehl gibt den Integerwert k zu der weiterführenden Code-Sequenz auf dem Return-Stack zurück.

Der A-Stack

Der A-Stack speichert Heap-Adressen. Erwartet eine STG-Funktion ein Argument vom A-Stack, dann erwartet sie einen Verweis auf einen Heap-Knoten, in dem der STG-Ausdruck für das Argument steht. Wir nennen diese Art von Argumenten auch *boxed values*.

Der B-Stack

Der B-Stack speichert Integerwerte. Eine STG-Funktion, die ihr Argument vom B-Stack bekommt, erhält keine Adresse, sondern den Wert des Argumentes selbst. Diese Art von Argumenten nennen wir *unboxed values*.

Der Return-Stack

Wenn eine *case*-Anweisung *case e of alts endcase* ausgeführt werden soll, so wird zunächst der Ausdruck e ausgewertet. Dann wird in Abhängigkeit des Ergebnisses eine Alternative der *alts* ausgeführt. Um die Alternativenliste *alts* vor der Auswertung des Ausdrucks zu speichern, gibt es den Return-Stack. Der Return-Stack speichert jedoch nicht nur die Alternativenliste, sondern auch die lokale Umgebung, ρ, in der später die ausgewählte Alternative ausgewertet werden soll.

Der Update-Stack

Der Update-Stack speichert Update-Informationen. Eine Update-Information ist dabei ein 3-tupel, (as_u, rs_u, a_u). as_u ist der alte Argument-Stack, rs_u ist der alte Return-Stack und a_u ist die Adresse des Heap-Knotens, der ausgewertet wird und der später aktualisiert werden soll. Bei dessen Auswertung werden der Argument-Stack und der Return-Stack durch neue, leere Stacks ersetzt. Nach der Auswertung werden die alten Stacks wiederhergestellt.

Der Heap

Ein Heap-Knoten hat die Form: $(vs \setminus\pi\ xs \rightarrow e)\ ws$. Die Lambdaform $(vs \setminus\pi\ xs \rightarrow e)$ stellt dabei den Code eines Heap-Knotens dar, während ws die Werte, die für die freien Variablen eines Heap-Knotens einzusetzen sind, angibt.

Die globale Umgebung

Die globale Umgebung gibt für globale Variablen, wie z.B. *main*, an, in welchem Heap-Knoten der zugehörige Code steht.

2.5.3 Die operationale Semantik

Die abstrakte STG-Maschine wird durch Zustandsübergangsregeln spezifiziert. Wir zeigen hier exemplarisch, wie das STG-Programm:

```
main = {} \u {} -> zweimal 2
zweimal = {} \n {B x} -> case x of Primitive
                          0 -> 0
                          y -> * {2,y}
                          endcase
```

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

mit

$$\sigma_{init} = \begin{bmatrix} main \rightarrow (Adr\,1) \\ zweimal \rightarrow (Adr\,2) \end{bmatrix}$$

$$h_{init} = \begin{bmatrix} 1 \rightarrow (\{\}\,\backslash u\,\{\} \rightarrow zweimal\{2\}) \\ 2 \rightarrow (\{\}\,\backslash n\,\{B\,x\} \rightarrow case...) \end{bmatrix}$$

Abbildung 2.5: Anfangszustand der abstrakten STG-Maschine

durch Anwendung der Zustandsübergangsregeln ausgewertet wird. Die abstrakte STG-Maschine beginnt die Auswertung im Startzustand. Dieser besteht aus der globalen Umgebung σ, die die globalen Variablen, *main* und *zweimal*, auf die Heap-Adressen 1 und 2 abbildet.

Im Heap stehen an den Adressen 1 und 2 die STG-Ausdrücke von *main* und *zweimal*. Der Argument-Stack, der Return-Stack und der Update-Stack sind zu Beginn leer.

Im Startzustand gibt die STG-Maschine den Befehl *Eval main* und beginnt damit ihre Arbeit (siehe Abbildung 2.5).

Die für die Anwendung von Variablen zuständige Regel der operationalen Semantik tritt dann in Kraft, die besagt, daß der STG-Befehl *Enter 1* ausgeführt werden soll. Damit wird der Heap-Knoten, der an Adresse 1 steht und der den STG-Ausdruck für *main* enthält, zur Auswertung gebracht. Die Information, daß der STG-Ausdruck für *main* an Adresse 1 steht, holt sich die STG-Maschine mit dem *val* Befehl, der in der globalen Umgebung die Heap-Adresse von *main* nachschlägt (siehe Abbildung 2.6).

Im Zustandsübergang für *Enter 1* wird der STG-Ausdruck, der im Heap an Adresse 1 steht, *zweimal* {*2*}, durch *Eval zweimal* {*2*} zur Auswertung gebracht.

Bei der Anwendung eines Heap-Knotens wird von der operationalen Semantik zuerst unterschieden, ob der Heap-Knoten aktualisiert werden soll, also ob sein *Update-Flag* gesetzt ist.

```
   Eval (main {}) {} {} {} {} {} h σ
mit
val ρ σ main = Adr 1

⟹Enter 1              {} {} {} {} h σ
```

Abbildung 2.6: Zustandsübergang für *Eval (main {})*

```
   Enter 1                {} {} {} {}  h σ
mit
h [1 → ({} \u {} → zweimal {2})]
⟹Eval zweimal {2} {}   {} {} {} us′  h σ


mit
us′ = ({},{},1)
```

Abbildung 2.7: Zustandsübergang für *Enter 1*

Da dies der Fall ist, wird der bisherige Argument-Stack, der bisherige Return-Stack, sowie die Adresse des zu aktualisierenden Heap-Knotens, als 3-tupel auf dem Update-Stack gespeichert. Der Argument-Stack und der Return-Stack werden durch leere Stacks ersetzt (siehe Abbildung 2.7).

Die Anweisung *Eval zweimal {2}* gibt an, daß die globale Variable *zweimal* auf das Argument 2 angewendet werden soll. Weil das Argument (die Zahl 2) ein Integerwert ist, wird dieses vor der Anwendung auf den B-Stack geschrieben. Wäre das Argument eine Adresse, so würden wir es auf den A-Stack schreiben. Dieses Vorgehen, die Argumente vor der Anwendung der Funktion auf die Argument-Stacks zu verteilen, nennen wir *tail-call* (siehe Abbildung 2.8).

Der Befehl *Enter 2* gibt an, daß der Inhalt des Heap-Knotens mit der Adresse 2 zur Auswertung gebracht werden soll. Dieser Heap-Knoten soll nicht aktualisert werden, deswegen bleibt der Update-Stack unverändert.

Eval (zweimal {2}) {} {} {}   {} ({},{},1) h σ
mit
val ρ σ zweimal = Adr 2

$\Longrightarrow$Enter 2   {} {2} {} ({},{},1) h σ

Abbildung 2.8: Zustandsübergang für *Eval (zweimal {2})*

Enter 2   {} {2} {} ({},{},1) h σ
mit
h [2 $\rightarrow$ ({} \n {B x} $\rightarrow$ case ...)]
$\Longrightarrow$Eval case ... ρ {} {}   {} ({},{},1) h σ

mit
ρ = [x $\rightarrow$ 2]

Abbildung 2.9: Zustandsübergang für *Enter 2*

Jetzt wird die lokale Umgebung ρ erzeugt, in der das Argument auf dem B-Stack (die Zahl 2) an die Variable des STG-Ausdrucks (x) gebunden wird.

Nun wird der STG-Ausdruck des Heap-Knotens, in diesem Fall: *case ...* mit *Eval case ...* zur Auswertung gebracht (siehe Abbildung 2.9).

Zur Auswertung des *case*-Ausdrucks werden die beiden Alternativen zusammen mit der lokalen Umgebung auf den Return-Stack geschrieben und der Ausdruck der *case*- Anweisung (x) wird durch *Eval* x ausgewertet.

Die Auswertung von *Eval* x, wobei x in der lokalen Umgebung an die Zahl 2 gebunden ist, führt zu dem STG-Befehl *ReturnInt 2*.

Der *ReturnInt* Befehl sucht sich nun, in Abhängigkeit seines Arguments die passende Alternative vom Return-Stack. Die erste Alternative mit dem *Tag* 0 paßt nur, wenn das Argument von *ReturnInt* die Zahl 0 ist. In allen anderen Fällen und auch in diesem Beispiel, wo das Argument von *ReturnInt* die Zahl 2 ist, wird die *Default-Alternative* ausgewählt.

Eval $*\{2,y\}$ ρ {} {} {} ({},{},1) h σ
mit
$\rho(y) = 2$
$\Longrightarrow$ReturnInt 4 {} {} {} ({},{},1) h σ

Abbildung 2.10: Zustandsübergang für *Eval * {2,y}*

Da die Default-Alternative eine Variable (y) mitangegeben hat, wird nun das Argument von *ReturnInt*, die Zahl 2, in der lokalen Umgebung an diese Variable gebunden und der Code der Alternative, in diesem Fall $*\{2,y\}$ wird durch *Eval* $*\{2,y\}$ zur Auswertung gebracht.

Die Auswertung des Ausdrucks $*\{2,y\}$ führt in der operationalen Semantik zur Berechnung von $2*y$ als eingebaute Operation und zur Rückgabe des Ergebnisses mit *ReturnInt 4*.

Wir sehen hier, warum die Multiplikation als *eingebaute Funktion* bezeichnet wird: Die abstrakte STG-Maschine führt die Multiplikation selbständig aus. Es macht dabei auch nichts aus, daß das zweite Argument (y) eine Variable ist. Die STG-Maschine holt sich den Wert der Variablen aus der lokalen Umgebung.

Nach der Auswertung wird die lokale Umgebung nicht mehr gebraucht und fallengelassen (siehe Abbildung 2.10).

Der Befehl *ReturnInt* könnte auch im Rahmen einer *case*-Anweisung abgesetzt worden sein. Deswegen prüft die STG-Maschine, sobald sie diesen Befehl bekommt, zuerst den Return-Stack auf evtl. vorhandende Alternativen von weiterführendem Code.

Da der Return-Stack leer ist, ist sicher, daß dieser *ReturnInt* Befehl nicht in Zusammenhang mit einer *case*-Anweisung steht und keine Alternativen-Liste nach passenden Alternativen durchsucht werden muß.

Es bleibt deshalb mit dem zurückgegebenen Integerwert nur noch eines zu tun: Die Ausgabe des Wertes[7].

[7] Eigentlich sollte nun eine Aktualisierung des Heap-Knotens mit der Adresse 1 stattfinden, da diese im Update-Stack gespeichert ist und die im Update-Stack gespeicherten Stacks sollten wiederhergestellt werden. Da dies aber keine funktionelle Änderung bringt, wurde hier darauf verzichtet.

Kapitel 3

Interpreter für die STG-Maschine

Wir stellen einen Interpreter für die STG-Maschine, wie sie in [PJ92] spezifiziert wird, vor.

Die Implementierung als Interpreter dient der Analyse des Ablaufs der STG-Maschine, da der Interpreter nach jedem Schritt den Zustand der STG-Maschine[1] ausgibt.

In dem o.g. Papier wird für die Implementierung der STG-Maschine die Verwendung von zwei Stacks, einen für *boxed values* und einen für *unboxed values* sowie die Arbeit mit *Return-Registern* vorgeschlagen.

Diese Methode ist zwar vermutlich Standard bei der Implementierung von funktionalen Sprachen und braucht nicht unbedingt auf die Ebene der abstrakten Maschine übernommen werden.

Wir verwenden in dem Interpreter aber zwei Stacks und Return-Register, damit die Auswertungsschritte des Interpreters ein Bild von der Arbeit der STG-Maschine vermitteln, das der Implementierung der STG-Maschine in Maschinensprache möglichst nahe kommt.

Die zwei Stacks treten an die Stelle des Argument-Stacks. Auf dem ersten Stack, dem sog. **A-Stack**, werden nur *boxed values*, also Adresswerte gespeichert. Der zweite Stack ist der **B-Stack**. Auf diesem werden nur *unboxed values*, also nur Integerwerte gespeichert.

Der Vorteil der Verwendung von *unboxed values* ist, daß wir Integerwerte nicht im Heap ablegen und dann ihre Adresse auf dem Argument-Stack speichern müssen. Wir können durch die Verwendung von *unboxed values* auf Integerwerte direk-

[1] Der Zustand der STG-Maschine setzt sich aus Ausgabe, Code, Argument-Stack, Return-Stack und Update-Stack zusammen.

ter zugreifen und müssen nicht für jeden Integerwert einen neuen Heap-Knoten erzeugen. Dies führt zu einer Effizienzsteigerung.

In der operationalen Semantik der STG-Maschine wird ein Rückgabewert einer Funktion, der ein Integerwert oder ein Konstruktor sein kann, nachdem er berechnet worden ist, als Argument von *ReturnInt*[2] bzw. *ReturnCon*[3] übergegeben.

Da der Integerwert bzw. der Konstruktor nach der Auswertung gleich wieder verwendet wird, ist es sinnvoll, wenn der Integerwert bzw. der Konstruktor mit seinen Argumenten in spezielle Register, die **Return-Register** geschrieben wird und die Befehle *ReturnInt* bzw. *ReturnCon* ohne Argumente aufgerufen werden.

Der Integerwert steht dann im ersten Return-Register. Bei der Übergabe von Konstruktoren steht der Konstruktor im ersten Return-Register und die Argumente stehen in den folgenden Return-Registern.

3.1 Beschreibung der Implementierung

3.1.1 Datenstrukturen für die STG-Programme

Die Datenstruktur für das STG-Programm, das wir interpretieren wollen, wird als eine Liste von *Bindings* implementiert.

```
> type StgProgram = Binds
> type Binds = [Bind]
```

Ein *Binding* ist dabei ein Paar aus einer Variablen und einer Lambdaform.

```
> type Bind = (Var,LambdaForm)
```

Der Parser übersetzt ein STG-Programm so, daß jede STG-Funktion einem *Binding* entspricht. Dabei wird der Name der STG-Funktion in *Var* und ihr Körper in *LambdaForm* gespeichert. *Var* ist ein String und benennt die Lambdaform. Die Lambdaform mit dem Namen *main* muß in jedem STG-Programm vorkommen. Die Lambdaform ist ein 4-tupel, bestehend aus den freien Variablen, dem Update-Flag, den Argumenten und einem STG-Ausdruck.

[2]STG-Maschinenbefehl zur Weiterverarbeitung von Integerwerten.
[3]STG-Maschinenbefehl zur Weiterverarbeitung von Konstruktoren.

```
> type Var = String
> type LambdaForm = (Vars,Update,Args,StgExpr)
```

Die freien Variablen sind Listen von Variablen:

```
> type Vars = [Var]
```

Das Update-Flag gibt an, ob die Repräsentation der Lambdaform im Heap überschrieben werden soll, wenn sie evaluiert wurde, oder nicht.

```
> data Update = Update | NoUpdate
```

Die Argumente sind eine Liste von Paaren, die aus der Information, auf welchem Stack ein Argument steht (A oder B) und den Variablen für das Argument bestehen.

```
> type Args = [Arg]
> type Arg = (Stack,Var)

> data Stack = A | B
```

Der STG-Ausdruck, gespeichert in *StgExpr*, kann eine der folgenden Formen annehmen:

```
> data STGExpr
>  = StgLet Binds StgExpr -- Let expressions
>  | StgLetrec Binds StgExpr -- Letrec expressions
>  | StgCase StgExpr Alts -- Case expressions
>  | StgApp Var Atoms -- Application
>  | StgCons Constr Atoms -- saturated Constructor
>  | StgBiOp Prim Atoms -- saturated built-in op
>  | StgLiteral Literal -- primitive Integer
```

Die STG-Ausdrücke *Let* und *Letrec* haben eine Liste von *Binds*, die die Variablen an die Werte binden und eine *StgExpr*, in der diese Variablen als lokale Variablen gültig sind.

Der *case*-Ausdruck enthält einen STG-Ausdruck, *StgExpr*, der zu einer Integerzahl oder zu einem Konstruktor ausgewertet wird. Ergibt der Ausdruck einen Konstruktor, so sind die Alternativen vom Typ *Algebraic*. Wird der Ausdruck zu einer Integerzahl ausgewertet, so sind die Alternativen vom Typ *Primitive*.

Alternativen vom Typ *Algebraic*

```
> data Alts = Algebraic [Aalt]
>            |Primitive [Palt]
```

Nach der Auswertung wird dann in der Alternativenliste *Aalt* für Konstruktoren und *Palt* für Integerwerte nachgeschaut, ob für diesen Konstruktor oder diese Integerzahl ein Eintrag vorhanden ist. Wenn ja, so werten wir den in diesem Eintrag stehenden STG-Ausdruck (*StgExpr*) aus; bei Konstruktoren weisen wir der in diesem Eintrag stehenden Variablenliste vor der Auswertung noch die Konstruktor-Argumente zu:

```
> type Aalt = AlgebraicAlt (Constr,Vars,StgExpr)
>           | AlgebraicDef Default

> type Palt = PrimitiveAlt (Literal,StgExpr)
>           | PrimitiveDef Default
```

Wurde der Konstruktor oder die Integerzahl nicht in der Liste gefunden, so wird die *Default-Alternative* angewendet. Diese hat entweder eine Variable mitangegeben, der nach der Auswertung des STG-Ausdrucks das Ergebnis zugewiesen wird (*DefaultVar*) oder der STG-Ausdruck wird nur ausgewertet, ohne daß das Ergebnis einer Variablen zugewiesen wird (*DefaultNoVar*):

```
> data Default = DefaultVar Var STGExpr
>               |DefaultNoVar STGExpr
```

Der STG-Ausdruck *STGApp* wendet eine Variable auf *Atoms* an. *Atoms* ist dabei eine Liste von *Atomen*, wobei ein *Atom* eine Integerzahl oder eine Variable sein kann:

```
> type Atoms = [Atom]
> data Atom = Literal Int | Variable Var
```

Ein Konstruktor besteht aus *Constr* (einer Integerzahl, die den Konstruktor identifiziert) und einer Liste von Atomen, auf die der Konstruktor angewendet wird.

```
> type Constr = Int
```

Der STG-Ausdruck *STGBiOp* steht für *Built-In-Operation*. Neben *Prim*, das die Operation spezifiziert, hat dieser Ausdruck noch eine Liste von Atomen als Argument. Es werden aber immer genau zwei Atome als Argument einer *Built-In-Operation* erwartet[4].

```
> data Prim = Plus  -- Addition
>           | Minus -- Subtraktion
>           | Mul   -- Multiplikation
>           | Div   -- Division
>           | Mod   -- Rest bei Division
>           | Eq    -- Gleichheit
>           | Le    -- Kleiner oder Gleich
>           | Lt    -- Kleiner
>           | Ge    -- Groesser oder Gleich
>           | Gt    -- Groesser
>           | Ne    -- Ungleicheit
>           | Sqrt  -- Wurzelfunktion
```

Schließlich gibt es als einfachsten STG-Ausdruck den Ausdruck, der nur aus einer Integerzahl besteht:

```
> type Literal = Int
```

[4]Bei *Sqrt*, der Wurzelfunktion und damit einzigen Operation, die nur ein Argument nimmt, wird nur das erste Argument verwendet.

3.1.2 Datenstrukturen für die STG-Maschine

Der STG-State

Die STG-Maschine besteht, wie schon beschrieben, aus den Komponenten *Code*, *Argument-Stack*, *Return-Stack*, *Heap* und *globaler Umgebung*. Zusätzlich müssen wir für die Implementierung eine *Output* Komponente und einen *Auswertungsstack* einführen. Alle diese Komponenten werden in dem benutzerdefinierten Datentyp *STGState* zusammengefaßt.

```
> type STGState = (StgOutput,
>                  StgEvalStack,
>                  StgCode,
>                  StgAs,
>                  StgBs,
>                  StgRs,
>                  StgUs,
>                  StgHeap,
>                  GlobalEnvm)
```

Da es in Gofer keine *records* gibt, sondern nur Tupel, müssen wir für den Zugriff auf die einzelnen Komponenten des *Stg-State* Zugriffsfunktionen vereinbaren, die alle den gleichen Aufbau haben.

Wir geben exemplarisch die Zugriffsfunktion, die die *Output-Komponente* aus dem *StgState* liefert, an:

```
> getOutput :: StgState -> StgOutput
> getOutput (out,es,code,as,bs,rs,us,heap,globals)
>  = out
```

Die Output-Komponente

Die *Output-Komponente* ist eine Zeichenliste, die jedesmal, wenn die STG-Maschine ein Zeichen ausgibt, um dieses Zeichen ergänzt wird.

```
> type STGOutput = [Char]
```

Der Eval-Stack

Lazy funktionale Sprachen, wie die STG-Sprache, werten Ausdrücke nur zur WHNF aus.

Dies bedeutet für das STG-Programm *inf*, das eine unendliche Liste von Zahlen erzeugt, daß es nur die erste Zahl der Liste berechnet und die Restliste als unausgewerteten Ausdruck zurückgibt.

```
inf = {} \n {B x} -> case + {x,1} of
                     Primitive y ->
                     let res = {} \n {}  -> inf y in <2> {x,res}
```

Zur kompletten Ausgabe muß *res* noch ausgewertet werden.

Dazu führen wir den Auswertungsstack ein, auf den Konstruktor- Argumente, wie *res*, geschrieben werden.

Die STG-Maschinen Anweisung *SEVAL* bewirkt, daß die Eintragungen des Auswertungsstacks ausgewertet werden.

In der Implementierung wird der Auswertungsstack durch eine Liste von Auswertungsstack-Einträgen (*ES*), wobei ein Eintrag ein *Value* oder ein *Klammer-zu* (*RP* für *Right Parenthesis*) Symbol sein kann, repräsentiert.

Das *Klammer-zu* Symbol wird benötigt, damit wir bei der Ausgabe der auf dem Auswertungsstack berechneten Werte durch Klammern deutlich machen können, zu welchem Konstruktor diese gehören.

```
> data ES = Entry Value | RP
> type STGEvalStack = [ES]
```

Ein *Value* kann entweder eine Heap-Adresse oder ein Integerwert sein:

```
> data Value = Adressvalue Adr | Integervalue Int
```

Die Code-Komponente

Der *Code* besteht aus einer Liste von Instruktionen, die bestimmen, wie ein STG-Programm ausgewertet werden soll. Beispielsweise muß zu Anfang immer die STG-Funktion *main* ausgewertet werden. Entsprechend startet die STG-Maschine mit der Instruktion *Eval main*.

Die weiteren Befehle, die schon beschrieben wurden, sind *Enter*, *ReturnCon*, *ReturnInt* und *SEval*.

```
> type STGCode = [Instruction]

> data Instruction = Eval STGExpr LocalEnvm
>                    |SEval LocalEnvm
>                    |Enter Adr LocalEnvm
>                    |ReturnCon Constr [Value] LocalEnvm
>                    |ReturnInt Int LocalEnvm
```

Der A-Stack

Der *A-Stack* beinhaltet die Argumente für eine Funktionsanwendung, die als *boxed values* vorliegen. Deswegen ist der A-Stack eine Liste von Adressen (*Adr*).

```
> type StgAs = [Adr]
```

Der B-Stack

Der *B-Stack* beinhaltet die Argumente für eine Funktionsanwendung, die als *unboxed values* vorliegen. Deswegen ist der B-Stack eine Liste von Integerwerten (*Int*).

```
> type StgBs = [Int]
```

Der Return-Stack

Der *Return-Stack* wird zur Auswertung der *case*-Anweisung benötigt. Er speichert ein 3-tupel, bestehend aus den Alternativen, zu denen nach der Auswertung des *case*-Ausdrucks verzweigt wird und den lokalen Umgebungen, die nach der Auswertung des *case*-Ausdrucks weiterverwendet werden sollen.

Solch ein 3-tupel nennen wir *Continuation*. Da es ggf. verschachtelte *case*-Anweisungen gibt, müssen wir in der Lage sein, uns mehrere *Continuations* zu merken. Deswegen ist der Return-Stack eine Liste von *Continuations*.

```
> type Continuation = (Alts,LocalEnvm1,LocalEnvm2)
> type STGRs = [Continuation]
```

Der Update-Stack

Der *Update-Stack* wird benötigt, um einen STG-Ausdruck auszuwerten, dessen *Update-Flag* gesetzt ist. Der Heap-Knoten, der den auszuwertenden STG-Ausdruck enthält, wird nach der Auswertung des Ausdrucks bei gesetztem *Update-Flag* mit dem Ergebnis überschrieben.

Bevor wir einen STG-Ausdruck, dessen *Update-Flag* gesetzt ist, auswerten, müssen wir A-Stack, B-Stack und Return-Stack speichern, da die Auswertung mit neuen (leeren) Stacks erfolgt, und die alten Stacks nach der Auswertung wiederhergestellt werden sollen.

Außerdem müssen wir uns die Adresse des Heap-Knotens, in dem der Ausdruck steht, merken, da der Heap-Knoten mit dem ausgewerteten Ausdruck überschrieben werden soll.

Wir speichern den A-Stack, B-Stack, Return-Stack und die Adresse des zu aktualisierenden Knotens in einem sog. *Updateframe*.

Weil es ggf. mehrere verschachtelte Ausdrücke gibt, die *updatable* sind und die ausgewertet werden müssen, müssen wir in der Lage sein, uns mehrere *Updateframes* zu merken. Deswegen ist der *Update-Stack* eine Liste von *Updateframes*.

```
> type Updateframe = (StgAs,StgBs,StgRs,Adr)
> type StgUs = [Updateframe]
```

Der Heap

Der *Heap* besteht aus einer Datenstruktur *Heap*, die die Adressverwaltung organisiert. Ein Heap-Knoten ist vom Typ *Closure*. Eine *Closure* speichert dabei jeweils eine Lambdaform und eine Liste von Werten, mit denen die freien Variablen dieser Lambdaform belegt sind.

```
> type STGHeap = Heap Closure

> data Closure = HeapItem LambdaForm [Value]
```

Die Globale Umgebung

Schließlich gibt es noch die *Global Environment*, die den Funktionsnamen die Heap-Adressen zuweist, an denen die zugehörigen Funktionskörper gespeichert sind.

```
> type GlobalEnvm = Assoc Var Adr
> type LocalEnvm  = Assoc Var Value
```

Kapitel 4

Optimierung der STG-Maschine

Abgesehen von den Return-Registern verwendet die STG-Maschine keine Register, sondern Stacks. Bei der Implementierung der STG-Maschine werden die Rechenoperationen (STGBiOp) jedoch durch Registeroperationen realisiert.

Wir versuchen, die STG-Maschine zu optimieren, indem wir die Register sichtbar und manipulierbar machen und uns damit stärker an der Hardware orientieren.

Zu diesem Zweck erweitern wir die STG-Sprache um explizite Registermanipulationsbefehle. Wir holen also die Registermanipulationsbefehle von der *Abstraktionsebene* der Maschinensprache auf die *Abstraktionsebene* der STG-Sprache herauf (siehe Abbildung 4.1).

Dazu verwenden wir zunächst abstrakte Register und abstrakte Registermanipulationsbefehle. Dies hat zum einen den Vorteil, daß der STG-Code mit Registermanipulationsbefehlen nicht maschinenabhängig ist.

Zum anderen kann der Kompiler[1] entscheiden, welche Hardware-Register er für die abstrakten Register verwendet. Wir können davon ausgehen, daß der Kompiler eine effiziente Registerbelegung erzielt.

4.1 Vorstellung des Optimierungsverfahrens

4.1.1 Die abstrakten Register und Registermanipulationsbefehle

Wir gehen von einer unbegrenzten Anzahl an abstrakten Registern aus. Hat die Zielmaschine weniger Register, so müssen einige Register auf Speicherzellen abgebildet werden.

[1] Hier ist der C-Kompiler gemeint. Wir übersetzen nämlich das STG-Programm mit abstrakten Registern in ein C-Programm mit abstrakten Registern und lassen dieses dann kompilieren.

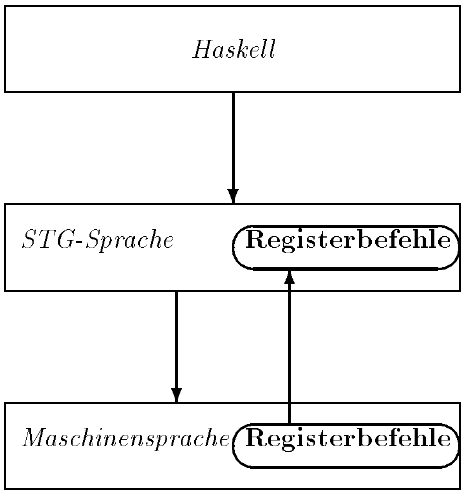

Abbildung 4.1: Registerbefehle in der STG-Sprache

In dieser Arbeit, in der wir als Zielmaschine die HP-PA RISC Architektur mit insgesamt 64 verfügbaren Registern verwenden, sollte dieses Problem jedoch nicht auftreten, zumal wir die Register nur zur Speicherung von Argumenten und lokalen Variablen benutzen werden.

Wir bezeichnen die abstrakten Register mit *arx*, wobei *x* die Register von 1 an durchnumeriert.

Die abstrakten Registerbefehle sind:

- **LOAD arx** *wert*: lädt das Register *arx* mit dem Wert *wert*.
- **ADD arx ary arz**: addiert den Inhalt von Register *arx* zu dem Inhalt von Register *ary* und schreibt das Ergebnis in Register *arz*.
- **SUB arx ary arz**: subtrahiert den Inhalt von Register *arx* vom Inhalt von Register *ary* und schreibt das Ergebnis in Register *arz*.
- **MUL arx ary arz**: multipliziert den Inhalt von Register *arx* mit dem Inhalt von Register *ary* und schreibt das Ergebnis in Register *arz*.
- **DIV arx ary arz**: dividiert den Inhalt von Register *arx* durch den Inhalt von Register *ary* und schreibt das Ergebnis in Register *arz*.
- **COPY arx ary**: kopiert den Inhalt von Register *arx* in Register *ary*.

STG-Ausdruck:	expr	$\rightarrow$ ...
		| "RE" rexpr
		| "RETURNINT"
Registerbefehle:	rexpr	$\rightarrow$ "LOAD" arx literal
		| "ADD" arx ary arz
		| "SUB" arx ary arz
		| "MUL" arx ary arz
		| "DIV" arx ary arz
		| "COPY" arx ary

Abbildung 4.2: Die Syntax der STG-Sprache mit Registermanipulationsbefehlen

4.1.2 Erweiterung der STG-Syntax

Die Syntax der STG-Sprache, wie sie in Abbildung 2.3 dargestellt ist, wird um die Möglichkeit, explizite Registermanipulationsbefehle auszudrücken, erweitert.

Dazu erweitern wir die Syntaxregel für *STG-Ausdruck* um die Möglichkeit *expr* $\rightarrow$ *RE rexpr*.

Unter *rexpr* werden dabei die Registermanipulationsbefehle gespeichert. Zusätzlich zu den Registermanipulationsbefehlen brauchen wir die Möglichkeit, die STG-Maschine anzuweisen, mit einem Wert, den wir mit Registermanipulationsbefehlen berechnet und in das Return-Register geschrieben haben, weiterzuarbeiten. Dazu muß die STG-Maschine mit *ReturnInt* weiterarbeiten; wir führen den Befehl *RETURNINT* ein, der die STG-Maschine dazu veranlaßt(siehe Abbildung 4.2).

Die Registerbefehle sind alle überladen. Wir können also mit *ADD ar1 1 ar10* die Zahl 1 auf Register *ar1* addieren und das Ergebnis in Register *ar10* schreiben. Die Überladungen werden vom Kompiler aufgelöst.

4.1.3 Die STG-Maschine mit expliziten Registermanipulationsbefehlen

Wir lassen die im letzten Abschnitt aufgezählten Registermanipulationsbefehle als imperative Befehle in STG-Funktionen zu. Verwendet eine STG-Funktion einen Registermanipulationsbefehl, so ist dieser Befehl auf ein bestimmtes Register festgelegt.

Für STG-Funktionen, die mit Registermanipulationsbefehlen arbeiten, erstellen wir zwei STG-Funktionen: Eine *Register-Push* Funktion, die die Werte in das richtige Register schreibt und eine *Core* Funktion, die mit den Werten in den Registern arbeitet.

Die *Register-Push* Funktion bekommt den Namen der ursprünglichen STG-Funktion und ersetzt diese damit.

Wir können auch die Return-Register direkt ansprechen. Wenn wir die Register ab *ar10* als Return-Register benutzen, dann sehen die STG-Funktionen, die unter Verwendung der Registermanipulationsbefehle zwei Werte addieren, so aus:

```
addiere = {} \n {B a,B b} ->
LOAD ar1 a       (lade ar1 mit a)
LOAD ar2 b       (lade ar2 mit b)
addiere_core {}  (rufe addiere_core auf)

addiere_core = {} \n {} ->
ADD ar1 ar2 ar10 (ar10 := ar1 + ar2)
RETURNINT        (fahre mit ReturnInt fort)
```

Die Funktion *addiere* schreibt die Argumente *a* und *b*, die auf dem B-Stack stehen, in die Register *ar1* und *ar2* und ruft dann die Funktion *addiere_core* auf, die keine Argumente bekommt.

Die Funktion *addiere_core* geht davon aus, daß in Register *ar1* und *ar2* die Argumente stehen und addiert diese. Das Ergebnis schreibt die Funktion in das Return-Register *ar10*.

Der Befehl *RETURNINT* wurde von der Ebene der STG-Maschine auf die Ebene der STG-Programme heraufgeholt. Er veranlaßt die STG-Maschine, mit *ReturnInt* fortzufahren.

Da *ReturnInt* bei leerem Return-Stack den Inhalt der Return-Register ausdruckt, wird in diesem Beispiel das Ergebnis ausgegeben.

Die Fakultätsfunktion sieht unter Verwendung der Registermanipulationsbefehle so aus:

```
fak = {} \n {B n} ->
LOAD ar1 n       (lade ar1 mit n)
fak_core {}      (rufe fak_core auf)
```

```
4  fak_core = {} \n {} ->
5  case
6  COPY ar1 ar10    (ar10 = ar1)
7  RETURNINT        (fahre mit ReturnInt fort)
8  of
9  1 -> 1
10 x ->
11 LOAD ar2 x       (ar2 = x)
12 SUB ar2 1 ar1    (ar1 = ar2 - 1)
13 fak_core {}      (rufe fak_core auf)
14 COPY ar10 ar4    (ar4 = ar10)
15 MUL ar1 ar4 ar10 (ar10 = ar1 + ar4)
16 RETURNINT
```

In Zeile 1 steht die *Register-Push* Funktion, *fak*. Diese schreibt in Zeile 2 ihr Argument, n, in Register 3 und ruft dann in Zeile 3 die *Core* Funktion (*fak_core*) auf.

In Zeile 4 beginnt die *Core* Funktion. Diese erwartet ihr Argument (n) in Register *ar1*.

Die *case*-Anweisung muß eine Fallunterscheidung nach dem Argument vornehmen. Da das Argument in Register *ar1* steht, wird dieses in Register *ar10*, das Return-Register, kopiert und mit *RETURNINT* wird die Selektion der passenden Alternative eingeleitet.

In Zeile 11 wird der Wert der Default-Variablen in Register *ar2* geladen, so daß wir über dieses Register Zugriff auf die Default-Variable haben.

In Zeile 16 wird in Register *ar1* der Wert *(n-1)* gebildet. Dann wird in Zeile 17 die *fak_core* Funktion direkt aufgerufen. Damit die Zeilen 14-16 nach der Abarbeitung von *fak_core* noch bearbeitet werden, schreiben wir diese als Default-Alternative auf den Return-Stack.

Die *fak_core* Funktion, die ihr Argument in Register *ar1* erwartet, berechnet jetzt die Fakultät von *(n-1)*. Das Ergebnis schreibt diese in das Return-Register *ar10*. Die STG-Maschine fährt nach der Abarbeitung der *fak_core* Funktion mit *ReturnInt* fort und führt so die als Default-Alternative auf dem Return-Stack stehenden Zeilen 14-16 aus.

In Zeilen 14 und 15 wird das Ergebnis des Aufrufs von *fak_core* (die Fakultät von *(n-1)*) mit n multipliziert.

In Zeile 16 wird der *RETURNINT* Befehl gegeben. Wenn wir die Rekursion beendet haben, ist der Return-Stack leer und das Ergebnis in Register *ar10* wird ausgegeben. Wenn wir uns noch in der Rekursion befinden, steht der noch auszuführende Code als Default-Alternative auf dem Return-Stack und wird ausgeführt.

4.1.4 Beurteilung der Optimierung

Wir stellen im folgenden zwei STG-Programme jeweils mit und ohne Verwendung von Registermanipulationsbefehlen gegenüber. Anhand der Anzahl der benötigten Registeroperationen beurteilen wir die Optimierung von STG-Programmen durch die Verwendung von expliziten Registermanipulationsbefehlen.

Das Haskell-Programm:

```
f a b c = (a+b) * c
```

wird in der STG-Maschine ohne Registermanipulationsbefehle zu folgendem STG-Code übersetzt:

```
f = {} \n {B a,B b,B c} -> case + {a,b} of
                            Primitive x -> * {x,c} endcase
```

Die STG-Maschine evaluiert[2] die Funktion *f*, indem sie a und b in Register schreibt (2 Registeroperationen), eine Addition ausführt und das Ergebnis in das Return-Register[3] überträgt (1 Registeroperation).

Dann bindet sie den Wert aus dem Return-Register an die lokale Variable x (1 Registeroperation), schreibt x und c wieder in Register (2 Registeroperationen) und führt eine Subtraktion aus. Das Ergebnis stellt die STG-Maschine wieder in das Return-Register (1 Registeroperation).

Insgesamt benötigen wir 7 Registeroperationen.

In der STG-Maschine mit Registermanipulationsbefehlen können wir die Tatsache, daß das Ergebnis der Addition mit in die Subtraktion eingeht, ausnutzen. Der STG-Code für die STG-Maschine mit Registermanipulationsbefehlen lautet:

[2] Entsprechend der Implementierung der STGBiOp Funktion.

[3] Mit *das Return-Register* ist hier und im folgenden das erste Return-Register, Register *ar10*, gemeint.

```
1 f = {} \n {B a,B b,B c} ->
2 LOAD ar1 a        (lade ar1 mit a)
3 LOAD ar2 b        (lade ar2 mit b)
4 LOAD ar3 c        (lade ar3 mit c)
5 f_core {}         (rufe f_core auf)

6 f_core = {} \n {} ->
7 ADD ar1 ar2 ar4   (ar4 = ar1 + ar2)
8 MUL ar4 ar3 ar10  (ar10 = ar4 * ar3)
9 RETURNINT         (fahre mit ReturnInt fort)
```

Die Funktion *f* überträgt in den Zeilen 2, 3 und 4 die Argumente *a*, *b* und *c* in die Register *ar1*, *ar2* und *ar3*. Dazu brauchen wir 3 Registeroperationen.

Dann wird in Zeile 5 die Funktion *f_core* aufgerufen.

Diese geht davon aus, daß die Argumente in Register *ar1*, *ar2* und *ar3* stehen. In Zeile 7 addiert diese den Inhalt von Register *ar1* zu dem Inhalt von Register *ar2* und schreibt das Ergebnis in Register *ar4* (1 Registeroperation).

In Zeile 8 multipliziert sie den Inhalt von Register *ar4* mit dem Inhalt von Register *ar3* und stellt das Ergebnis in das Return-Register *ar10* (1 Registeroperation).

Schließlich wird mit dem *RETURNINT* Befehl in Zeile 9 die STG-Maschine angewiesen, mit *ReturnInt* fortzufahren.

Wir brauchen jetzt nur 5 Registeroperationen. Zwei Registeroperationen konnten wir sparen, da durch die Verwendung der Registermanipulationsbefehle das Umkopieren des Zwischenergebnisses *(a+b)* überflüssig wurde.

Optimierung durch Core nach Core-Aufruf

Im folgendem Beispiel ruft die STG-Funktion *binkoff* die STG-Funktion *fak* auf.

```
binkoff = {} \n {B n, B k} ->
case fak{n} of
  Primitive n1 -> case fak{k} of
    Primitive k1 -> case -{n,k} of
      Primitive nk -> case fak{nk} of
        Primitive nk1 -> case *{k1,nk1} of
          Primitive k1nk1 -> /{n1,k1nk1}
        endcase
```

```
9         endcase
10      endcase
11    endcase
12 endcase
```

In Zeile 2 wird die Fakultätsfunktion mit dem Argument n aufgerufen. Diese schreibt den Wert n in ein Register (1 Registeroperation) und berechnet dann die Fakultät.

Das Endergebnis überträgt die Fakultätsfunktion in das Return-Register (s Registeroperationen[4]).

Der Wert aus dem Return-Register wird in Zeile 3 an die lokale Variable *n1* gebunden (1 Registeroperation). Dann wird die Fakultätsfunktion erneut aufgerufen, die den Wert k in ein Register schreibt (1 Registeroperation) und dann die Fakultät berechnet und das Ergebnis in das Return-Register stellt (s Registeroperationen).

In Zeile 4 wird der Wert aus dem Return-Register an die lokale Variable *k1* gebunden (1 Registeroperation). Zur Auswertung von $-\{n, k\}$ werden die Variablen n und k in Register geschrieben (2 Registeroperationen). Dann wird die Subtraktion ausgeführt und das Ergebnis wird in das Return-Register übertragen (1 Registeroperation).

In Zeile 5 wird der Wert aus dem Return-Register an die lokale Variable *nk* gebunden (1 Registeroperation). Nach Aufruf der Fakultätsfunktion wird der Wert von *nk* einem Register zugewiesen (1 Registeroperation), die Fakultät wird berechnet und das Ergebnis in das Return-Register geschrieben (s Registeroperationen).

In Zeile 6 wird der Wert aus dem Return-Register an die lokale Variable *nk1* gebunden (1 Registeroperation). Dann werden die Werte *k1* und *nk1* in Register geschrieben (2 Registeroperationen). Die Multiplikation wird ausgeführt und das Ergebnis in das Return-Register gestellt (1 Registeroperation).

In Zeile 7 wird der Inhalt des Return-Registers an die lokale Variable *k1nk1* gebunden (1 Registeroperation). Dann werden die Werte *n1* und *k1nk1* in Register geladen (2 Registeroperationen) und die Division wird ausgeführt, die das Ergebnis in das Return-Register schreibt (1 Registeroperation).

Es wurden **insgesamt** $17 + 3s$ **Registeroperationen** ausgeführt. Aus der Ablaufbeschreibung wird deutlich, daß ein großer Teil dieser Registeroperationen unnötig ist, da nur Zwischenergebnisse umkopiert werden.

[4]Wir interessieren uns für die Anzahl der von der *binkoff* Funktion benötigten Registeroperationen, jeweils mit und ohne Verwendung der Registermanipulationsbefehle. Die Anzahl an Registeroperationen, die die Fakultätsfunktion braucht, wird hier sowohl mit als auch ohne Verwendung der Registermanipulationsbefehle mit s abgeschätzt.

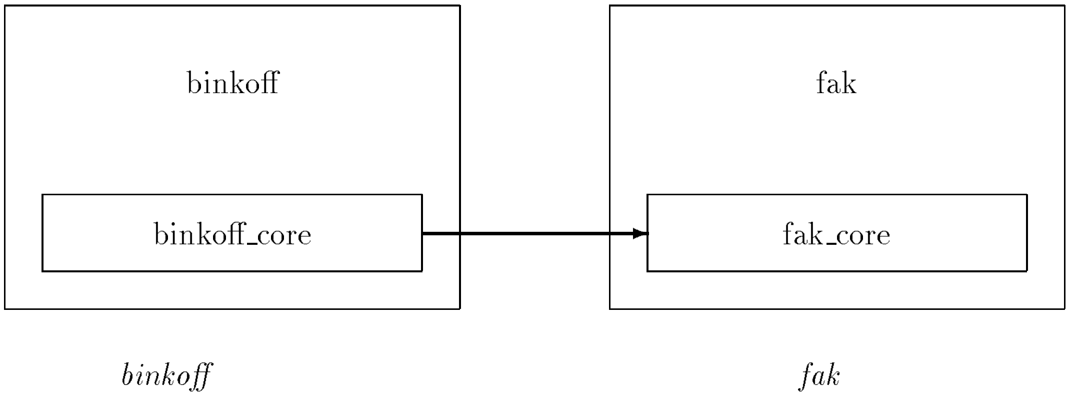

Abbildung 4.3: Die *Core* Funktion von *binkoff* ruft direkt die *Core* Funktion von *fak* auf.

Durch die Verwendung von Registermanipulationsbefehlen können wir die unnötigen Registeroperationen vermeiden. Neben der Ersparnis an Registeroperationen für das Umkopieren der Zwischenergebnisse können wir noch die Registeroperationen sparen, die bei der Übergabe der Argumente beim Aufruf der Fakultätsfunktion benötigt werden.

Dies geschieht dadurch, daß wir von der *binkoff_core* Funktion aus direkt die *fak_core* Funktion aufrufen (siehe Abbildung 4.3) und vorher die Argumente für die Fakultätsfunktion in die Register schreiben, in denen die *fak_core* Funktion ihre Argumente erwartet:

```
binkoff = {} \n {B n,B k} ->
LOAD ar1 n          (lade ar1 mit n)
LOAD ar2 k          (lade ar2 mit k)
binkoff_core {}     (rufe binkoff_core auf)

binkoff_core = {} \n {} ->
SUB ar1 ar2 ar3     (r3 = r1 - r2)
fak_core {}         (rufe fak_core auf)
COPY ar10 ar4       (ar4 = ar10)
COPY ar2 ar1        (ar2 = ar1)
fak_core            (rufe fak_core auf)
COPY ar10 ar5       (ar5 = ar10)
COPY ar3 ar1        (ar1 = ar3)
fak_core            (rufe fak_core auf)
```

```
14 MUL ar5 ar10 ar6   (ar6 = ar5 * ar10)
15 DIV ar4 ar6 ar10   (ar10 = ar4 / ar6)
16 RETURNINT          (fahre mit ReturnInt fort)
```

In Zeile 2 und 3 werden die Werte n und k in Register geladen (2 Registeroperationen).

In Zeile 4 wird die *binkoff_core* Funktion aufgerufen. In Register *ar1* steht jetzt der Wert n und in Register *ar2* der Wert k.

In Zeile 6 wird Register *ar1* (Wert n) von Register *ar2* (Wert k) abgezogen und das Ergebnis $(n-k)$ wird in Register *ar3* geschrieben (1 Registeroperation).

In Zeile 7 wird die Fakultäts-*Core* Funktion aufgerufen. Diese arbeitet mit dem Wert, der in Register *ar1* steht *(n)* und schreibt das Ergebnis *(n!)* in Register *ar10*, das Return-Register (s Registeroperationen).

In Zeile 8 wird der Wert von Register *ar10 (n!)* in Register *ar4* geschrieben (1 Registeroperation).

In Zeile 9 wird der Wert von Register *ar2 (k)* nach Register *ar1* verschoben (1 Registeroperation). In Register *ar4* steht jetzt *n!* und in Register *ar1* der Wert k.

In Zeile 10 wird die Fakultäts-*Core* Funktion aufgerufen. Da in Register *ar1* der Wert k steht, berechnet diese nun *k!* und schreibt dieses Ergebnis in Register *ar10* (s Registeroperationen).

In Zeile 11 wird der Inhalt von Register *ar10 (k!)* nach Register *ar5* verschoben. In Register *ar4* steht jetzt der Wert *n!* und in Register *ar5* steht der Wert *k!* (1 Registeroperation).

In Zeile 12 wird der Wert aus Register *ar3* $(n-k)$ in Register *ar1* geschoben (1 Registeroperation).

In Zeile 13 wird die Fakultäts-*Core* Funktion aufgerufen, die $(n-k)!$ berechnet. Sie schreibt dieses Ergebnis in Register *ar10* (s Registeroperationen).

In Zeile 14 wird Register *ar5 (k!)* mit Register *ar10 ((n-k)!)* multipliziert und das Ergebnis $(k! * (n-k)!)$ wird in Register *ar6* geschrieben (1 Registeroperation).

In Zeile 15 wird der Wert in Register *ar4 (n!)* durch den Wert in Register *ar6* $(k! * (n-k)!)$ dividiert. Das Ergebnis ist das Endergebnis $(n!/(k! * (n-k)!))$. Dieses wird in Register *ar10* geschrieben (1 Registeroperation).

Wir benötigten **insgesamt** $10 + 3s$ **Registeroperationen**. Das sind 7 weniger als das STG-Programm ohne Registermanipulationsbefehle benötigte.

Dabei konnten wir die Registeroperationen sparen, die wir vorher für das Umkopieren der Argumente für die Fakultätsfunktion und der Zwischenergebnisse benötigten.

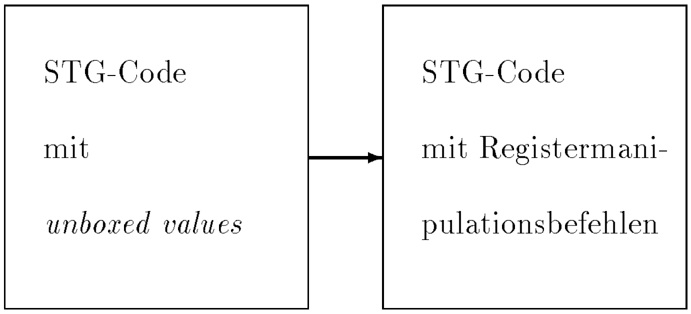

Abbildung 4.4: Die Übersetzung von STG-Code in STG-Code mit Registermanipulationsbefehlen.

Wir sind also in der Lage durch Zulassung von expliziten Registermanipulationsbefehlen in STG-Programmen die Anzahl der benötigten Registeroperationen zu minimieren.

4.1.5 Erzeugung von STG-Code mit expliziten Registermanipulationsbefehlen

Wir können alle STG-Funktionen, die ausschließlich mit *unboxed values* arbeiten, durch STG-Funktionen, die Registermanipulationsbefehle verwenden, ersetzen.

Die Verwendung von Registermanipulationsbefehlen bewirkt eine Auswertung aller Ausdrücke. Funktionen, die ausschließlich mit *unboxed values* arbeiten, sind *strikt*. In diesen Funktionen werden auch ohne Registermanipulationsbefehle alle Ausdrücke ausgewertet.

In Funktionen, die auch mit *boxed values* arbeiten, werden Ausdrücke, deren Wert nicht benötigt wird, nicht ausgewertet. Durch die Verwendung von Registermanipulationsbefehlen in Funktionen, die auch mit *boxed values* arbeiten, würden evtl. Ausdrücke ausgewertet werden, deren Wert gar nicht gebraucht wird und die Sprache wäre nicht mehr *lazy*.

Wir gehen also von STG-Code mit der Information, welche Werte *boxed* und welche Werte *unboxed* sind, aus und erzeugen STG-Code mit Registermanipulationsbefehlen (siehe Abbildung 4.4). Dieser STG-Code mit Registermanipulationsbefehlen wird (über den Zwischenschritt der Übersetzung in C) in Assembler übersetzt.

$\mathbf{f} = vars_f \ \backslash\pi \ \{\mathbf{B}\ arg_1,\ldots,\mathbf{B}\ arg_n\} \rightarrow expr$
$\longrightarrow$
$\mathbf{f}\ vars_f \ \backslash\pi \ \{\mathbf{B}\ arg_1,\ldots,\mathbf{B}\ arg_n\} \rightarrow$
LOAD ar1 arg_1
LOAD ar2 arg_2
...
f_core

f_core {} \ n {} $\rightarrow$
Regel2 (*expr*)

Abbildung 4.5: Regel 1

Jede STG-Funktion, die ausschließlich mit *unboxed values* arbeitet, wird durch zwei STG-Funktionen mit Registermanipulationsbefehlen ersetzt: Die *Register-Push* Funktion und die *Core* Funktion.

Die *Register-Push* Funktion lädt die Argumente in die Register und ruft dann die *Core* Funktion auf. Da die *Register-Push* Funktion genauso heißt, wie die ursprüngliche STG-Funktion, die gleichen Argumente nimmt und das gleiche Ergebnis liefert, können Aufrufe der ursprünglichen STG-Funktion im STG-Code unverändert stehen bleiben.

Die *Core* Funktion geht davon aus, daß alle Argumente in den Registern stehen. Sie führt die Berechnungen unter Verwendung der Registermanipulationsbefehle durch, schreibt das Ergebnis in das Return-Register und gibt den *RETURN-INT* Befehl, der die STG-Maschine veranlaßt, mit dem Ergebnis in dem Return-Register weiterzuarbeiten.

Der Code für die *Register-Push* Funktion ist einfach: Die *Register-Push* Funktion nimmt die gleichen Argumente wie die STG-Funktion ohne Registermanipulationsbefehle und schreibt diese alle in die Register.

Dabei wird das erste Argument (*arg*) mit *LOAD arx arg* in das nächste freie abstrakte Register *arx* geschrieben, das zweite in Register *ar(x+1)* usw. Dann ruft die *Register-Push* Funktion die *Core* Funktion auf. Die *Core* Funktion erhält keine Argumente, da diese schon in den Registern stehen.

Regel 1 (siehe Abbildung 4.5) erzeugt für eine beliebige Funktion *f* die *Register-Push* Funktion und den Kopf der *Core* Funktion. Der Funktionskörper der *Core* Funktion wird von Regel 2 (siehe Abbildung 4.6) erzeugt, die von Regel 1 mit dem Argument *expr* aufgerufen wird.

Im ersten Schritt von Regel 2 werden Zwischenergebnisse, die durch die *case-*

- **case** *expr1* **of**
  alts; *defaultvar* → *expr2*
  ⟶
  case Regel2(*expr1*) **of** *alts'*
  defaultvar →
  LOAD arx defaultvar
  Regel2(*expr2*)
  mit:
  arx = nächstes freies Register
  alts' = sukzessive Anwendung von Regel 2 auf die Ausdrücke von *alts*.

- + {vx,vy}
  mit: vx steht in arx, vy steht in ary
  ⟶
  ADD arx ary ar10
  RETURNINT

- - {vx,vy}
  mit: vx steht in arx, vy steht in ary
  ⟶
  SUB arx ary ar10
  RETURNINT

- * {vx,vy}
  mit: vx steht in arx, vy steht in ary
  ⟶
  MUL arx ary ar10
  RETURNINT

- / {vx,vy}
  mit: vx steht in arx, vy steht in ary
  ⟶
  DIV arx ary ar10
  RETURNINT

- **expr**
  ⟶
  expr

Abbildung 4.6: Regel 2

Anweisung an eine lokale Variablen gebunden werden, in ein eigenes Register kopiert.

Steht das Zwischenergebnis in der lokalen Variablen *vx*, so wird dieses durch *LOAD vx arx* in Register *arx* kopiert. Das Register *arx* ist dabei ein Register, das noch nicht durch die *Register-Push* Funktion zur Speicherung eines Argumentes verwendet wurde und in dem Regel 2 noch keine andere lokale Variable speichert.

In den Ersetzungsanweisungen für +, -, * und / sind die *vx* und *vy* Variablen, die für Argumente der Funktion stehen oder lokale Variablen. Ist eine Variable ein Argument der Funktion, so wurde der Wert der Variablen von der *Register-Push* Funktion in einem bestimmten Register *arx*, bzw. *ary* gespeichert. Ist eine Variable eine lokale Variable, so wurde diese Variable durch die *case*-Anweisung gebunden und durch den Ersetzungschritt für *case* in Regel 2 wurde der Wert der lokalen Variablen in das Register *arx* bzw. *ary* geschrieben.

Bei den Ersetzungsanweisungen für +, -, * und / müssen wir außerdem noch den *RETURNINT* Befehl geben, da in diesen Fällen etwas in das Return-Register geschrieben wurde und die STG-Maschine diesen Wert mit *ReturnInt* weiterverarbeiten muß.

Die oben angegebene Funktion *f* in STG-Sprache ohne Registermanipulationbefehle:

```
f = {} \n {B a,B b,B c} -> case + {a,b} of
                           Primitive x -> * {x,c} endcase
```

wird durch Anwendung von Regel 1 und Regel 2 in folgende zwei STG-Funktionen mit Registermanipulationsbefehlen übersetzt:

```
f = {} \n {B a,B b,B c} ->
LOAD ar1 a          (lade ar1 mit a)
LOAD ar2 b          (lade ar2 mit b)
LOAD ar3 c          (lade ar3 mit c)
f_core {}           (rufe f_core auf)

f_core = {} \n {} ->
case
  ADD ar1 ar1 ar10  (ar10 = ar1 + ar2)
  RETURNINT         (fahre mit ReturnInt fort)
  of x ->
  LOAD ar4 x        (lade ar4 mit x)
```

```
  MUL ar4 ar3 ar10  (ar10 = ar4 * ar3)
  RETURNINT         (fahre mit ReturnInt fort)
 endcase
```

Die oben angegebene Funktion *binkoff* in STG-Sprache ohne Registermanipulationbefehle:

```
binkoff = {} \n {B n, B k} ->
case fak{n} of
  n1 -> case fak{k} of
    k1 -> case -{n,k} of
      nk -> case fak{nk} of
        nk1 -> case *{k1,nk1} of
          k1nk1 -> /{n1,k1nk1}
        endcase
      endcase
    endcase
  endcase
endcase
```

wird durch Anwendung von Regel 1 und Regel 2 in folgende zwei STG-Funktionen mit Registermanipulationsbefehlen übersetzt:

```
binkoff = {} \n {B n, B k} ->
LOAD ar1 n                  (lade ar1 mit n)
LOAD ar2 k                  (lade ar2 mit k)
binkoff_core {}             (rufe binkoff_core auf)

binkoff_core = {} \n {} ->
case fak{n} of
  n1 ->
  LOAD ar3 n1               (lade ar3 mit n1)
  case fak{k} of
    k1 ->
    LOAD ar4 k1             (lade ar4 mit k1)
    case
      SUB ar1 ar2 ar10      (ar10 = ar1 - ar2)
      RETURNINT             (fahre mit ReturnInt fort)
```

```
      of nk ->
      LOAD ar5 nk          (lade ar5 mit nk)
      case
        fak{nk} of
        nk1 ->
        LOAD ar6 nk1       (lade ar6 mit nk1)
        case
          MUL ar4 ar6 ra10 (ar10 = ar4 * ar6)
          RETURNINT        (fahre mit ReturnInt fort)
          of k1nk1 ->
          LOAD ar7 k1nk1   (lade ar7 mit k1nk1)
          DIV ar3 ar7 ar10 (ar10 = ar3 / ar7)
          RETURNINT        (fahre mit ReturnInt fort)
          endcase
        endcase
      endcase
    endcase
  endcase
endcase
```

Beide STG-Funktionen verwenden durch die Konvertierung Registermanipulationsbefehle, doch ist die Anzahl der benötigten Registeroperationen gegenüber den STG-Funktionen ohne Registermanipulationsbefehle nicht geringer, da die Zwischenergebnisse immer noch in lokalen Variablen gespeichert werden.

4.1.6 Optimierung des erzeugten STG-Codes mit Registermanipulationsbefehlen

Vorbereitungsschritt: case-Anweisungen mit nur einer Alternative

Wurde im STG-Code ohne Registermanipulationsbefehle die *case*- Anweisung mit nur einer Alternative benutzt, so wurde diese verwendet, um die Auswertung des Ausdrucks hinter der *case*-Anweisung zu erzwingen. Das Ergebnis der Auswertung wurde dann einer lokalen Variablen zugewiesen.

Da die Verwendung von Registermanipulationsbefehlen alleine schon die Auswertung erzwingt, können wir die *case*- Anweisungen mit nur einer Alternative bei der Verwendung von Registermanipulationsbefehlen weglassen.

Wir können auch darauf verzichten, das Ergebnis der Auswertung einer lokalen Variablen zuzuweisen, wenn wir dieses Ergebnis in einem Register speichern. Dies geschieht dadurch, daß wir das Zwischenergebnis, das sich immer im Return-Register *ar10* befindet, mit dem *COPY* Befehl in ein Register verschieben. Wir nehmen dafür das Register, dem wir das Zwischenergebnis in Regel 2 mit LOAD zugewiesen haben.

Dieses Register darf im weiteren Verlauf der Funktion nicht überschrieben werden: das Zwischenergebnis muß, wie wenn wir es in einer lokalen Variablen speichern, über den ganzen weiteren Ablauf der Funktion verfügbar bleiben.

Anstelle einer lokalen Variablen verwenden wir dann immer das entsprechende Register. Dadurch ändern sich auch die Registerbefehle: Haben wir vorher mit *LOAD arx va* die lokale Variable *va* in Register *arx* geladen, so verwenden wir jetzt *COPY ara arx* wobei die Variable *va* im Register *ara* gespeichert ist.

Wir dürfen die *case*-Anweisung natürlich nur weglassen, wenn die Auswertung des Ausdrucks durch die Verwendung von Registermanipulationsbefehlen erzwungen wird.

Dies ist der Fall, wenn der Ausdruck aus einer Rechenoperation (+, -, *, /) oder dem Aufruf einer Funktion, die nur *unboxed* Argumente bekommt[5], besteht.

Dieser Vorbereitungsschritt ist in Regel 3 angegeben (siehe Abbildung 4.7)

Wenden wir Regel 3 auf den STG-Code mit Registermanipulationsbefehlen an, den wir mit Regel 1 und Regel 2 für die Funktion *f* erzeugt haben, so ergibt sich folgender STG-Code[6]:

```
f = {} \n {B a,B b,B c} ->
LOAD ar1 a      (lade ar1 mit a)
LOAD ar2 b      (lade ar2 mit b)
LOAD ar3 c      (lade ar3 mit c)
f_core {}       (rufe f_core auf)

f_core = {} \n {} ->
ADD ar1 ar2 r10  (ar10 = ar1 + ar2)
* COPY ar10 ar4  (ar4 = ar10)
MUL ar4 ar3 ar10 (ar10 = ar4 * ar3)
RETURNINT        (fahre mit ReturnInt fort)
```

Wenden wir Regel 3 auf den STG-Code mit Registermanipulationsbefehlen an, den wir mit Regel 1 und Regel 2 für die Funktion *binkoff* erzeugt haben, so ergibt sich folgender STG-Code:

[5] Funktionen, die nur *unboxed* Argumente bekommen, werden in Funktionen mit Registermanipulationsbefehlen übersetzt.

[6] Die Zeilen, die sich geändert haben, werden hier und im folgenden mit (*) gekennzeichnet.

- **case** $\oplus$ ara arb ar10
  RETURNINT
  of
  defaultvar $\rightarrow$ **LOAD** arz defaultvar
  expr2
  $\longrightarrow$
  $\oplus$ ara arb ar10
  COPY ar10 arz
  expr2'
  mit:
  expr2' = *expr2*
  aber statt: LOAD ary *defaultvar*
  $\rightarrow$ COPY arz ary (wenn arz = ary entfällt die Anweisung).
  $\oplus \in$ { **ADD, SUB, MUL, DIV** }

- **case f** (B arg_1,...,B arg_n) **of**
  defaultvar $\rightarrow$ **LOAD** arz defaultvar
  expr2
  $\longrightarrow$
  f (B arg_1,...,B arg_n)
  COPY ar10 arz
  expr2' mit:
  expr2' = *expr2*
  aber statt: LOAD ary *defaultvar*
  $\rightarrow$ COPY arz ary (wenn arz = ary entfällt die Anweisung).

Abbildung 4.7: Regel 3

```
binkoff = {} \n {B n, B k} ->
LOAD ar1 n          (lade ar1 mit n)
LOAD ar2 k          (lade ar2 mit k)
binkoff_core {}     (rufe binkoff_core auf)

binkoff_core = {} \n {} ->
* fak{n}
* COPY ar10 ar3       (ar3 statt lokale Variable n1)
* fak{k}
* COPY ar10 ar4       (ar4 statt lokale Variable k1)
* SUB ar1 ar2 ar10    (ar10 = ar1 - ar2)
* COPY ar10 ar5       (ar5 statt lokale Variable nk)
* fak{nk}
* COPY ar10 ar6       (ar6 statt lokale Variable nk1)
* MUL ar4 ar6 ar10    (ar10 = ar4 * ar6)
* COPY ar10 ar7       (ar7 statt lokale Variable k1nk1)
* DIV ar3 ar7 ar10    (ar10 = ar3 / ar7)
RETURNINT           (fahre mit ReturnInt fort)
```

Optimierungsschritt: Direktaufruf von *Core* Funktionen

Wenn eine beliebige Funktion *f* aufgerufen wird, von der eine *Core* Funktion existiert, dann können wir die *Core* Funktion direkt aufrufen, wenn wir vorher die Argumente von *f* in die Register schreiben, in denen die *Core* Funktion diese erwartet.

Das erste Argument erwartet jede *Core* Funktion in Register *ar1*, das zweite in Register *ar2* usw.

Dieser Optimierungsschritt wird in Regel 4 (siehe Abbildung 4.8) dargestellt.

Die Anwendung von Regel 4 auf den STG-Code für die Funktion *f* bringt keine Änderung, da *f* keine Funktionen aufruft. Wenn wir Regel 4 auf den STG-Code für die *binkoff* Funktion anwenden, auf den wir schon Regel 1 bis 3 angewendet haben, so ergibt sich folgender STG-Code:

```
binkoff = {} \n {B n, B k} ->
LOAD ar1 n              (lade ar1 mit n)
LOAD ar2 k              (lade ar2 mit k)
binkoff_core {}         (rufe binkoff_core auf)
```

- **f** (**B** arg_1,...,**B** arg_n)
  mit:
  f_core existiert
  $\longrightarrow$
  COPY arx ar1, mit: arg_1 steht in arx
  COPY ary ar2, mit: arg_2 steht in ary
  ...
  f_core {}

Abbildung 4.8: Regel 4

```
binkoff_core = {} \n {} ->
* fak_core {}         (rufe fak_core mit Argument in ar1 auf)
COPY ar10 ar3         (ar3 statt lokale Variable n1)
* COPY ar2 ar1        (fak wird mit arg. k aufgerufen)
* fak_core {}         (rufe fak_work auf)
COPY ar10 ar4         (ar4 statt lokale Variable k1)
SUB ar1 ar2 ar10      (ar10 = ar1 - ar2)
COPY ar10 ar5         (ar5 statt lokale Variable nk)
* COPY ar5 ar1        (fak wird mit Arg. nk aufgerufen)
* fak_core {}         (rufe fak_core auf)
COPY ar10 ar6         (ar6 statt lokale Variable nk1)
MUL ar4 ar6 ar10      (ar10 = ar4 * ar6)
COPY ar10 ar7         (ar7 statt lokale Variable k1nk1)
DIV ar3 ar7 ar10      (ar10 = ar3 / ar7)
RETURNINT             (fahre mit ReturnInt fort)
```

Der STG-Code mit Registermanipulationsbefehlen für die *binkoff* Funktion entspricht dem im Abschnitt *Beurteilung der Optimierung* vorgestellten optimalen STG-Code mit Registermanipulationsbefehlen.

Der STG-Code mit Registermanipulationsbefehlen für die Funktion *f* wird im nächsten Schritt noch verbessert.

Nachbereitungsschritt: Zusammenfassen von Befehlen

Wir ersetzen die Kombination der Befehle: $\oplus$ *arx ary arz* und *COPY arz ara* durch die Anweisung, die das gleiche Resultat direkt erreicht: $\oplus$ *rx ry ra*. ($\oplus$

- $\oplus$ **arx ary arz** und **COPY arz ara**
  $\longrightarrow$
  $\oplus$ **arx ary ara**
  mit:
  $\oplus \in$ { **ADD, SUB, MUL, DIV** }

Abbildung 4.9: Regel 5

steht hier wieder für ADD, SUB, DIV oder MUL).

Den Nachbereitungsschritt stellt Regel 5 dar (siehe Abbildung 4.9).

Wir wenden Regel 5 auf den STG-Code für die Funktion *f* an, auf den wir schon Regel 1 bis 4 angewendet haben:

```
f = {} \n {B a,B b,B c} ->
LOAD ar1 a        (lade ar1 mit a)
LOAD ar2 b        (lade ar2 mit b)
LOAD ar3 c        (lade ar3 mit c)
f_core {}         (rufe f_core auf)

f_core = {} \n {} ->
* ADD ar1 ar2 ar4 (ar4 = ar1 + ar2)
MUL ar4 ar3 ar10  (r10 = ar4 * ar3)
RETURNINT         (fahre mit ReturnInt fort)
```

Wir sehen, daß der im Abschnitt *Beurteilung der Optimierung* schon vorgestellte optimale STG-Code mit Registermanipulationsbefehlen erzeugt wurde.

4.2 Implementierung des Optimierungsverfahrens

4.2.1 Die Filter-Funktion

Die Filter-Funktion bekommt als Eingabe ein STG-Programm ohne Registermanipulationsbefehle und liefert als Ausgabe das STG-Programm mit Registermanipulationsbefehlen, das durch die Anwendung der Regeln 1 bis 5 erzeugt wurde.

Für die Filter-Funktion haben wir die STG-Syntax wie folgt erweitert:

```
data StgExpr
= ...
| CAT StgExpr StgExpr
| RE RExpr
| RETURNINT
| EMPTY

data RExpr
= LOAD Var Var
| ADD Var Var Var
| SUB Var VAr Var
| MUL Var VAr Var
| DIV Var VAr Var
| COPY Var VAr
```

RE RExpr, wobei *RExpr* aus *LOAD, ADD, SUB, MUL, DIV oder COPY* bestehen kann und *RETURNINT* wurden schon besprochen.

Neu sind *CAT StgExpr StgExpr* und *EMPTY*.

CAT StgExpr StgExpr wird benötigt, um mehrere aufeinanderfolgende Registerausdrücke in einem STG-Programm darstellen zu können.

EMPTY brauchen wir in der Filter-Funktion, wenn wir unnötige Befehle, wie z.B. *COPY ar10 ar10*, weglassen wollen.

Die Filter-Funktion bekommt als Eingabe eine Integerzahl, sowie das STG-Programm ohne Registermanipulationsbefehle und liefert als Ausgabe das STG-Programm mit Registermanipulationsbefehlen:

```
fil :: Int -> StgProgram -> StgProgramm
```

Die Integerzahl kann die Werte von 0 bis 3 annehmen: Bei 0 wendet die Filter-Funktion nur die Regeln 1 und 2 an, d.h. es wird nur STG-Code mit Registermanipulationsbefehlen erzeugt, ohne daß dieser optimiert wird. Für 1 wendet die Filter-Funktion zusätzlich zu Regeln 1 und 2 noch Regel 3 an, d.h. unnötige *case*-Anweisungen werden wegoptimiert. Wird die Integerzahl auf 2 gesetzt, werden Regeln 1-4 angewendet, d.h. die Optimierung von *Core* zu *Core* Aufrufen wird durchgeführt. Wenn die Integerzahl auf 3 gesetzt ist, werden alle Optimierungsregeln, also auch noch Regel 5, die unnötige Befehle wegläßt, angewendet.

Da die Filter-Funktion durch die angegebenen Regeln 1 bis 5 schon spezifiziert ist, verzichten wir hier auf den Abdruck des weiteren Programms.

4.2.2 Vorstellung der Zielarchitektur

Die Zielarchitektur ist die HP PA RISC 1.1 Architektur der HP 9000 Series 600/700/800 Computer, die an unserem Fachbereich eingesetzt werden. PA bedeutet dabei Precision Architektur. Die hier vorgestellten Daten wurden aus dem *Assembly Language Reference Manual* [Hew91] entnommen.

Der HP PA RISC Prozessor hat 4 Registerbänke:

- 32 allgemeine Register, je 32 Bit breit, %r0 bis %r31. Register %r0, %r1 sowie Register %r19 bis %r31 sind nicht benutzbar, da sie für interne Informationen verwendet werden. Es bleiben die 16 Register %r3 bis %r18.
- 32 Fließkomma-Register, %fr0 bis %fr31. Register %fr0 bis %fr7 sind dem Programmierer nicht zugänglich. Es bleiben die 24 Register %fr8 bis %fr31. Benutzen wir die Fließkomma-Register nicht, um doppelt-genaue IEEE Zahlen zu speichern, sondern genügen auch einfach-genaue IEEE Zahlen, so können wir mit %fr0L bis %fr31L und %fr0R bis %fr31R auch die linke bzw. rechte Hälfte der Register als Register für einfach-genaue IEEE Zahlen verwenden. Da uns für unsere Anwendung einfach-genaue IEEE Zahlen ausreichen, haben wir nochmals 48 Register und damit insgesamt 64 Register zur Verfügung.
- 8 Speicher-Register, %sr0 bis %sr7, die als Basis für den virtuellen Speicher dienen und von denen wir deshalb keinen Gebrauch machen können.
- 25 Kontroll-Register, %cr0 und %cr8 bis %cr31. Diese Register enthalten Systeminformationen und sind deshalb auch nicht verfügbar.

Die wichtigsten Registermanipulationsbefehle sind:

- *FDIV register1 register2 register3*:
  Fließkommadivision
- *FMPY register1 register2 register3*:
  Fließkommamultiplikation
- *FSUB register1 register2 register3*:
  Fließkommasubtraktion

- *FADD register1 register2 register3*:
  Fließkommaaddition

- *FCPY quellregister zielregister*:
  Fließkommakopieren

- *LDI integerwert zielregister*:
  Laden des Zielregisters mit dem Integerwert

- *COMIBT integerwert register offset*:
  Vergleicht den Integerwert mit dem Wert in dem Register und springt bei Gleichheit um *Offset* Zellen weiter.

- *COMIBF integerwert register offset*:
  Vergleicht den Integerwert mit dem Wert in dem Register und springt bei Ungleichheit um *Offset* Zellen weiter.

4.2.3 Abbildung der abstrakten Register auf die Zielarchiktektur

Wir gehen von STG-Code mit Registermanipulationsbefehlen für die abstrakten Register aus und wollen Maschinencode mit Registermanipulationsbefehlen für die HP PA RISC 1.1 Architektur erzeugen.

Dazu verwenden wir den Kompiler von Frank Kiesel, um den STG-Code mit Registermanipulationsbefehlen für die abstrakten Register in C-Code mit Registermanipulationsbefehlen für die abstrakten Register zu übersetzen.

Im C-Code bilden wir dabei die abstrakten Register auf globale Variablen ab und weisen den C-Kompiler an, daß diese in Hardware-Registern gehalten werden sollen.

Der C-Kompiler erzeugt dann den Maschinencode für die HP PA RISC 1.1 Architektur mit Registermanipulationsbefehlen (siehe Abbildung 4.10).

Um den STG-Code mit Registermanipulationsbefehlen für die abstrakten Register in C-Code mit Registermanipulationsbefehlen für die abstrakten Register zu übersetzen, erweitern wir den Kompiler von Frank Kiesel um folgende Regeln:

- Das abstrakte Register *arx* im STG-Code übersetzen wir in die globale Variable *arx* im C-Code und geben die Kompiler-Anweisung, diese Variable in einem Register zu halten.

- **LOAD arx** *wert* im STG-Code übersetzen wir in die C-Anweisung *arx = wert*.

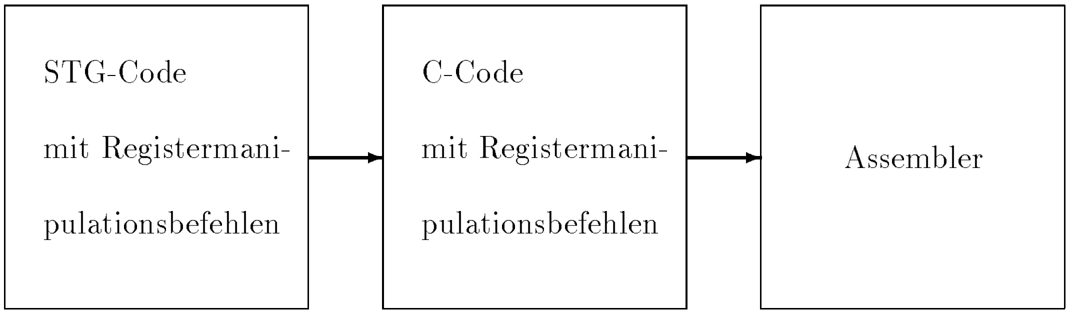

Abbildung 4.10: Die Übersetzung von STG-Code mit Registermanipulationsbefehlen in Maschinencode.

- **ADD arx ary arz** im STG-Code übersetzen wir in die C-Anweisung *arz = arx + ary.*
- **SUB arx ary arz** im STG-Code übersetzen wir in die C-Anweisung *arz = arx - ary.*
- **MUL arx ary arz** im STG-Code übersetzen wir in die C-Anweisung *arz = arx * ary.*
- **DIV arx ary arz** im STG-Code übersetzen wir in die C-Anweisung *arz = arx / ary.*
- **COPY arx ary** im STG-Code übersetzen wir in die C-Anweisung *ary = arx.*

Der C-Code mit Registermanipulationsbefehlen kann mit einem C-Kompiler in Assembler übersetzt werden, wobei der Kompiler die abstrakten Register auf Hardware-Register abbildet.

4.3 Resultate

Um die Optimierung zu beurteilen, vergleichen wir folgendes STG-Fakultätsprogramm mit dem STG-Programm mit Registermanipulationsbefehlen, das die Filter-Funktion daraus erzeugt hat.

STG-Programm ohne Registermanipulationsbefehle:

```
fak = {} \n  {B:n,B:ak}  -> case n  of
      1 -> ak
      m -> case * {n,ak} of
                  ak2 -> case - {n,1} of
                  n1 -> fak n1,ak2
```

STG-Programm mit Registermanipulationsbefehlen, erzeugt von der Filter-Funktion:

```
fak = {} \n  {B:x,B:ak}  -> LOAD ar1 x
                            LOAD ar2 ak
                            fak_core {}

fak_core = {} \n  {}  -> case x  of
    1 -> ak
    z -> LOAD ar3 z
         case SUB ar1 1 ar10
              RETURNINT of
                 x1 -> LOAD ar4 x1
                       case MUL ar1 ar2 ar10
                            RETURNINT of
                              ak2 -> LOAD ar5 ak2
                                     fak x1,ak2
```

4.3.1 Vergleich des resultierenden C- und Assemblercodes

Wir lassen beide STG-Programme von dem Kompiler von Frank Kiesel, den wir um Registermanipulationsbefehle erweitert haben, in C-Code übersetzen. Die entsprechenden C-Programme befinden sich in Anhang B.

Im C-Programm für das STG-Programm ohne Registermanipulationsbefehle ruft sich die Funktion *fak_direct* selbst auf (Zeile 15). Vorher wird das Argument auf den B-Stack geschrieben (Zeile 14).

Im C-Programm für das STG-Programm mit Registermanipulationsbefehlen ruft sich die Funktion *fak_core_direct* selbst auf (Zeile 17). Das Argument wird vorher in das abstrakte Register *ar1* geschrieben (Zeile 16).

Architektur	Registermanipulationsbefehle	Zeit
PC	ohne	5.66s
PC	mit	5.06s

Abbildung 4.11: Ausführungszeiten für das kompilierte STG-Fakultätsprogramm auf dem PC

Die uneffiziente Argumentübergabe über den B-Stack fällt also zu Gunsten der effizienten Argumentübergabe über abstrakte Register weg.

Im Anhang B befindet sich auch der Assemblercode, der durch die Kompilierung der beiden C-Programme entsteht.

Das Assemblerprogramm für das STG-Programm ohne Registermanipulationsbefehle legt in Zeile 70 das Argument für die Fakultätsfunktion, die sich in Zeile 76 selbst aufruft, auf den B-Stack.

Das Assemblerprogramm für das STG-Programm mit Registermanipulationsbefehlen schreibt in Zeile 51 das Argument für die Fakultätsfunktion, die sich in Zeile 55 selbst aufruft, in das abstrakte Register *ar1*, das auf ein Hardware-Register abgebildet wurde.

Auch hier fällt die uneffiziente Argumentübergabe über den B-Stack zu Gunsten der effizienten Argumentübergabe über abstrakte Register weg.

4.3.2 Vergleich der Ausführungszeiten

In der Tabelle in Abbildung 4.11 sind die Ausführungszeiten für das Maschinenspracheprogramm jeweils für das STG-Programm mit und ohne Registermanipulationsbefehlen auf einem PC angegeben.

Da die Zeiten für die Ausführung des Programms so klein waren, daß wir sie nicht messen konnten, führten wir das Fakultätsprogramm jeweils 1000 mal hintereinander aus. Wir berechneten die Fakultät von 20.

Die Ausführungszeit für den PC (386, 8MB, kein Coproz.) zeigt den erwarteten Gewinn in dem zu erwartendem Umfang.

Auf der HP PA RISC Architektur konnten wir an der Ausführungszeit jedoch keinen Vorteil der Optimierung feststellen.

Wenn wir von den Ausführungszeiten auf dem PC ausgehen, so konnten wir eine Verbesserung von etwa 10 Prozent erreichen.

Das Fakultätsprogramm ruft sich 19 mal rekursiv auf. D.h. 19 mal wird das Argument anstatt über den B-Stack über Register übergeben. Da wir das Fakultätsprogramm 1000 mal hintereinander ausgeführt haben, beträgt die Anzahl der Schritte, in denen Argumente anstatt über den B-Stack über Register übergeben werden 19.000.

Dividieren wir die Zeit von 0.6 Sekunden, die das Programm mit der Argumentübergabe über Register schneller läuft durch die 19.000 so können wir die gewonnene Zeit für die Übergabe eines Argumentes über Register anstelle des B-Stacks mit 0.000 031 579 Sekunden angeben.

Verglichen mit dem (geringen) Aufwand, der nötig ist, die Optimierung durchzuführen, nämlich die Filter-Funktion auf das STG-Programm anzuwenden, lohnt sich die Optimierung.

Kapitel 5

Konzeptueller Vergleich STG/G-Maschine

5.1 Flache Applikationen

5.1.1 Was sind flache Applikationen ?

Unter *flachen Applikationen* verstehen wir, daß Funktionen nur auf Konstanten oder Variablen angewendet werden dürfen. Z.B. sind die Anwendung einer Funktion auf eine andere Funktion, wie $f(g(x))$ oder die Anwendung einer Funktion auf einen Ausdruck, wie $f(x+1)$ keine flachen Applikationen.

5.1.2 Flache Applikationen bei der STG-Maschine

Flache Applikationen sind in der STG-Maschine zwingend vorgeschrieben. Dies liegt darin begründet, daß die STG-Maschine den STG-Code direkt auswertet und nicht umstellt.

Wäre z.B. $f(x + 1)$ in einem STG-Programm erlaubt, so müsste die STG-Maschine, wenn sie auf f stößt, die Auswertung von f zurückstellen und erst $x + 1$ auswerten. Dazu müsste aber der STG-Code umgestellt werden, was nicht geht.

Wenn wir einen Ausdruck wie $f(x + 1)$ in der STG-Sprache formulieren wollen, so muß erst $x + 1$ mit einem *let*-Ausdruck gebunden werden und dann f auf das Ergebnis angewendet werden.

5.1.3 Flache Applikationen bei der G-Maschine

Die G-Maschine schreibt flache Applikationen nicht vor. In der G-Maschine können wir also Funktionen auf andere Funktionen und auf Ausdrücke anwenden.

Wir werden zeigen, daß die G-Maschine intern trotzdem mit flachen Applikationen arbeitet.

Dazu lassen wir zunächst ein Core-Language Programm, daß nicht-flache Applikationen verwendet, durch die G-Maschine ausführen.

Dann transformieren wir das Core-Language Programm so, daß es flache Applikationen verwendet und lassen es erneut von der G-Maschine ausführen.

Core-Language Programm ohne flache Applikationen:

```
main = f(f(f(f(f(1)))))

f(x) = x + 1
```

G-Code für *main*:	G-Code für *f*:
Pushint 1	Pushint 1
Pushglobal f	Push 1
Mkap	Eval
Pushglobal f	Add
Mkap	Update 1
Pushglobal f	Pop 1
Mkap	Unwind
Pushglobal f	
Mkap	
Pushglobal f	
Mkap	
Eval	
Update 0	
Pop 0	
Unwind	

Laufzeit:
Steps taken = 71
(119445 reductions, 337200 cells, 5 garbage collections)

Core-Language Programm mit flachen Applikationen[1]:

[1] Dieses Core-Language Programm entspricht vom Aufbau her dem STG-Code.

```
main = let f1 = f(1) in let f2 = f(f1) in let f3 = f(f2)
       in let f4 = f(f3) in f(f4)

f(x) = x + 1
```

G-Code für *main*:	G-Code für *f*:
Pushint 1	Pushint 1
Pushglobal f	Push 1
Mkap	Eval
Push 0	Add
Pushglobal f	Update 1
Mkap	Pop 1
Push 0	Unwind
Pushglobal f	
Mkap	
Push 0	
Pushglobal f	
Mkap	
Push 0	
Pushglobal f	
Mkap	
Eval	
Slide 1	
Slide 1	
Slide 1	
Slide 1	
Update 0	
Pop 0	
Unwind	

Laufzeit:
Steps taken = 79
(148739 reductions, 415758 cells, 7 garbage collections)

Wenn wir den G-Code mit und ohne flache Applikationen vergleichen, so sehen wir, daß er, bis auf einige unnötige *Push* und *Slide* Anweisungen, derselbe ist.

Der G-Code ohne flache Applikationen braucht 8 Schritte weniger, weil dieser Code die direkte Applikation von *f* auf mehrere Argumente ist und der G-Code mit flachen Applikationen die Argumente erst mit dem *Push* Befehl auf den Stack legt und dann mit *Slide*[2] wieder beseitigt.

[2] Die 4 aufeinanderfolgenden *Slide 1* Anweisungen lassen sich vom Kompiler zu einer *Slide*

Wir können als Ergebnis festhalten: Auch die G-Maschine arbeitet intern mit flachen Applikationen.

5.2 Updates in Place

5.2.1 Was sind Updates in Place?

Funktionen werden bei abstrakten Maschinen in Heap-Knoten gespeichert. Nimmt eine Funktion keine Argumente, so liefert sie immer das gleiche Ergebnis, egal in welchem Kontext sie aufgerufen wird.

Wenn wir eine Funktion, die keine Argumente nimmt, das erste Mal ausgewertet haben, so ist es sinnvoll, wenn wir uns das Ergebnis merken, um die Funktion bei möglichen wiederholten Aufrufen nicht erneut auswerten zu müssen.

Eine Möglichkeit, uns das Ergebnis einer bereits ausgewerteten Funktion ohne Argumente zu merken ist es, eine *Indirektion* von dem Funktionsnamen zu der Stelle, an der das Ergebnis steht, zeigen zu lassen. Dieses Konzept wird bei der G-Maschine verwendet. Wir nennen diese Methode *Update durch Indirektionsknoten*.

Eine andere Möglichkeit, das Ergebnis einer ausgewerteten Funktion verfügbar zu halten, ist es, die Berechnungsvorschrift, also die *rechte Seite* der Funktion mit dem Ergebnis zu überschreiben[3].

Diese Möglichkeit verwendet die STG-Maschine. Wir nennen diese Methode *Update in Place*.

5.2.2 Updates in Place bei der STG-Maschine

Wie erwähnt verwendet die STG-Maschine *Updates in Place*. Ein Heap-Knoten, der eine Funktion ohne Argumente speichert, wird bei der STG-Maschine jedoch nur dann mit dem Ergebnis überschrieben, wenn dies im STG-Code angegeben ist.

---

4 Anweisung zusammenfassen. Der G-Code, der für das Core-Language Programm mit der Verwendung von flachen Applikationen erzeugt wurde, hat dann aber immer noch 5 (unnötige) Anweisungen mehr als der G-Code, der für das Core-Programm ohne Verwendung von flachen Applikationen erzeugt wurde.

[3] Bei dieser zweiten Möglichkeit müssen wir die abstrakte Maschine so implementieren, daß alle Heap-Knoten gleich groß sind, denn wir überschreiben hier einen Heap-Knoten mit einem anderen. Im Umfang dieser Arbeit tritt dieses Problem nicht auf, weil wir die Interpreter für die abstrakten Maschinen in Gofer schreiben und den Heap als abstrakte Datenstruktur von Gofer verwalten lassen.

Dazu verfügt bei der STG-Maschine jede Repräsentation einer Funktion über ein *Update-Flag*. Nur wenn das Update-Flag gesetzt ist, wird der Heap-Knoten, der die Funktion speichert, nach der Auswertung mit dem Ergebnis überschrieben.

Dies hat den Vorteil, daß sich die STG-Maschine den Schritt, den Heap-Knoten der Funktion zu überschreiben, für die Fälle spart, in denen das Ergebnis der Funktion während der Programmauswertung nur einmal gebraucht wird.

5.2.3 Versuch der Optimierung der G-Maschine durch Updates in Place

Die G-Maschine arbeitet mit *Updates durch Indirektionsknoten*. Durch den G-Befehl *Update n* wird der Heap-Knoten, auf den der *n*. Stack-Eintrag verweist[4], mit dem Inhalt des Heap-Knotens aktualisiert, auf den der oberste Stack-Eintrag verweist.

Der zu aktualisierende Heap-Knoten wird dabei durch einen *Indirektionsknoten* ersetzt, der auf den Heap-Knoten verweist, mit dem er aktualisiert werden soll.

Wir wollen bei der G-Maschine *Updates in Place* einführen, wie sie auch von der STG-Maschine benutzt werden.

Bei einem *Update in Place* wird der zu aktualisierende Heap-Knoten direkt mit dem Inhalt des Heap-Knotens ersetzt, mit dem er aktualisiert werden soll.

Um zu untersuchen, ob wir durch die Einführung von *Updates in Place* bei der G-Maschine eine Verbesserung erzielen, lassen wir zunächst das Core-Language Programm: **main = 1 + 2** von der G-Maschine mit *Updates durch Indirektionsknoten* ausführen.

Dieses Programm wird zu folgender G-Code Sequenz übersetzt:

```
Code for main
  Code:{Pushint 2
        Pushint 1
        Add
        Update 0
        Pop 0
        Unwind}
```

Interessant ist für uns hierbei der G-Befehl *Update 0*, der einen *Update durch Indirektionsknoten* ausführt.

Die G-Maschine beginnt die Ausführung, indem sie die Befehle *Pushglobal main* und *Eval*, die die *main* Funktion auswerten, und den Befehl *Print*, der das Ergebnis ausgibt, zur Ausführung bringt:

[4] Der oberste Stack-Eintrag zählt nicht mit, der zweitoberste hat die Nummer 0 usw.

```
1) Output://
    Stack:[]
    Dump:[]
     Code:{Pushglobal main
           Eval
           Print}
```

Im nächsten Schritt steht auf dem Stack die Adresse 7, die auf den Heap-Knoten, der *main* enthält, verweist und der *Eval* Befehl steht zur Ausführung an:

```
2) Output://
    Stack:[7: Global main]
    Dump:[]
     Code:{Eval
           Print}
```

In den Schritten 3 bis 6 wurde die Addition ausgeführt. Nach der Ausführung der Addition steht das Ergebnis auf der obersten Stackposition, in diesem Fall die Heap-Adresse 22, die den Integerwert 3 enthält.

Mit dem Befehl *Update 0* soll der Heap-Knoten, auf den der 0.Stack-Eintrag verweist[5] mit dem Heap-Knoten, auf den der oberste Stack-Eintrag verweist[6], aktualisiert werden.

```
7) Output://
    Stack:[7: Global main
           22: 3]
    Dump:[<{Print}, []>]
     Code:{Update 0
           Pop 0
           Unwind}
```

Im nächsten Schritt hat die G-Maschine eine Aktualisierung durch Indirektionsknoten vorgenommen: Der Heap-Knoten mit der Adresse 7 wurde mit einem Indirektionsknoten überschrieben, der auf die Heap-Adresse 22 zeigt.

[5]In diesem Fall der Heap-Knoten mit der Adresse 7, der *main* enthält.

[6]In diesem Fall der Heap-Knoten mit der Adresse 22, der den Integerwert 3 enthält.

```
 8) Output://
     Stack:[7: Ind 22]
     Dump:[<{Print}, []>]
      Code:{Pop 0
            Unwind}
```

```
 9) Output://
     Stack:[7: Ind 22]
     Dump:[<{Print}, []>]
      Code:{Unwind}
```

Der Befehl *Pop 0* hat keine Funktion, da 0 Elemente vom Stack entfernt werden. Der Befehl *Unwind* löst schließlich die Indirektion auf, indem der Heap-Knoten 7 mit dem Inhalt des Heap-Knotens ersetzt wird, auf den die Indirektion verweist:

```
10) Output://
     Stack:[22: 3]
     Dump:[<{Print}, []>]
      Code:{Unwind}
```

Zum Schluß wird durch den *Print* Befehl das Ergebnis ausgegeben:

```
11) Output://
     Stack:[22: 3]
     Dump:[]
      Code:{Print}
```

```
12) Output:/3/
     Stack:[]
     Dump:[]
      Code:{}
```

Die Statistik über die Reduktionen und benutzten Zellen, die Gofer jetzt ausgibt, verwenden wir später im Vergleich mit der G-Maschine mit *Updates in Place*:

```
Steps taken = 11
(18578 reductions, 52290 cells, 1 garbage collection)
```

G-Maschine mit Updates in Place

Wir führen jetzt *Updates in Place* bei der G-Maschine ein. Dazu definieren wir einen neuen G-Befehl: *Updateip n*, der wie der *Update* Befehl den Heap-Knoten aktualisiert, auf den der n.Stack-Eintrag verweist.

Der *Updateip* Befehl stellt jedoch keine Indirektion her, sondern überschreibt den Heap-Knoten, der aktualisiert werden soll, mit dem Inhalt des Heap-Knotens, auf den der oberste Stack-Eintrag verweist.

Wir lassen wieder das Core-Language Programm: **main = 1 + 2** auf der G-Maschine ausführen. In der G-Code Sequenz verwenden wir diesmal jedoch nicht den *Update* Befehl, sondern den *Updateip* Befehl:

```
Code for main
  Code:{Pushint 2
        Pushint 1
        Add
        Updateip 0
        Pop 0
        Unwind}
```

Die Schritte 1 bis 6 laufen wie bei der G-Maschine mit *Updates durch Indirektionsknoten* ab. In Schritt 7 steht an der obersten Stackposition wieder die Adresse des Heap-Knotens mit dem Ergebnis[7] und an der Stackposition 0 steht die Adresse des Heap-Knotens, der *main* enthält.

Die G-Maschine mit *Updates durch Indirektionsknoten* führte in Schritt 7 durch den *Update* Befehl einen *Update durch Indirektionsknoten* aus.

Die G-Maschine mit *Updates in Place* gibt jetzt den Befehl *Updateip*, d.h. der Inhalt des Heap-Knotens mit der Adresse 7 soll mit dem Inhalt des Heap-Knotens mit der Adresse 22 überschrieben werden:

```
   7) Output://
```

[7] Das Ergebnis ist der Integerwert 3, der in der Heap-Adresse 22 steht.

```
      Stack:[7: Global main
             22: 3]
      Dump:[<{Print}, []>]
       Code:{Updateip 0
             Pop 0
             Unwind}
```

Wie wir im nächsten Schritt sehen, wird der *Update in Place* richtig ausgeführt:

```
   8) Output://
      Stack:[7: 3]
      Dump:[<{Print}, []>]
       Code:{Pop 0
             Unwind}
```

Schließlich wird in Schritt 11 das Ergebnis ausgegeben:

```
  11) Output:/3/
      Stack:[]
      Dump:[]
       Code:{}
```

Die G-Maschine mit *Updates in Place* spart sich den Schritt, den die G-Maschine mit *Updates durch Indirektionsknoten* braucht, um die Indirektion aufzulösen[8].

Deswegen kommt die G-Maschine mit *Updates in Place* auch mit weniger Reduktionen aus:

```
Steps taken = 10
(18067 reductions, 50847 cells, 1 garbage collection)
```

Als Resultat können wir festhalten, daß *Updates in Place* eine Effizienzsteigerung bei der G-Maschine bewirken, da wir uns die Schritte sparen können, die Indirektionsknoten wieder aufzulösen.

Außerdem haben *Updates in Place* den Vorteil, daß zur Ausführung weniger Heap-Knoten benötigt werden. Es kommt daher bei der G-Maschine, die mit *Updates*

[8]Schritt 9 bei der G-Maschine mit Updates durch Indirektionsknoten.

in Place arbeitet erst später zur *Garbage Collection*, einem Prozeß, mit dem Heap-Knoten, die nicht mehr benutzt werden, wieder zur Benutzung freigegeben werden.

Wir können die G-Maschine also durch *Updates in Place* bezüglich Zeit- und Platzbedarf optimieren[9].

5.3 Übersetzung von STG-Code in G-Code

5.3.1 Können wir STG-Code direkt in G-Code übersetzen?

Hierzu überlegen wir uns, wie die in der STG-Sprache möglichen Ausdrücke in G-Code aussehen würden:

Als erstes betrachten wir die STG-Anweisung **main = <literal>**, die einer Funktion als Ergebnis einen Integerwert zuweist. Als korrelierender G-Code ergibt sich:

```
Pushint <literal>
Update 0
Pop 0
Unwind
```

Die STG-Anweisung **main = <Variable> <Atom1>**, die die Funktionsapplikation einer Variablen auf ein Atom darstellt, läßt sich folgendermaßen als G-Code ausdrücken:

```
Pushint <Atom1>
Pushglobal <Variable>
Mkap
Eval
Update 0
Pop 0
Unwind
```

[9]Simon L. Peyton Jones schlägt in [PJL91], Abschnitt 2.5.1 *Reducing the number of indirections*, ein ähnliches Vorgehen vor: Wenn ein Knoten, auf den eine Indirektion verweisen soll, neu erzeugt werden muß, dann wird anstelle einer Indirektion ein *Update in Place* ausgeführt. Wurde dieses Vorgehen in der Implementierung berücksichtigt, dann bedeutet die Einführung von *Updates in Place* bei der G-Maschine keine Optimierung: Wenn Simon L. Peyton Jones kein *Update in Place* ausführt, muß er zwar eine Indirektion auflösen, dafür muß er aber keinen neuen Heap-Knoten erzeugen.

Die Anwendung eines Konstruktors ohne Argumente, also die STG-Anweisung **main = <Constr>** {} ergibt folgenden G-Code:

```
Pack 1 0
Update 0
Pop 0
Unwind
```

Die STG-Anweisung **main = <BiOp>** {**<Atom1>,<Atom2>**}, die die Anwendung einer *Built-In Operation* auf zwei Atome darstellt, läßt sich direkt in G-Code übersetzen, wobei zu beachten ist, daß der G-Code Befehl <BiOp> auch in der G-Maschine eine *Built-In Operation* ist, also ein Befehl, den die G-Maschine direkt ausführt und um den wir uns nicht weiter kümmern brauchen:

```
Pushint <Atom1>
Pushint <Atom2>
<BiOp>
Update 0
Pop 0
Unwind
```

Waren bisher alle STG-Anweisungen relativ direkt in G-Code zu übersetzen, so erfordet die *case*-Anweisung in der STG-Sprache eine schon aufmerksamere Betrachtung:

Für jede Alternative wird nämlich hinter dem G-Code Befehl *Casejump* ein *Tag* mit zugehörigem Code-Segment erzeugt. Die STG-Anweisung **main = case <expr1> of Primitive 1** $\rightarrow$ **<expr2> endcase** entspricht dem G-Code:

```
  Eval
  Casejump 1  Code:{Split0
                    <expr2>
                    Slide 0}
  Update 0
  Pop 0
  Unwind
```

Die STG-Anweisung **letrec <expr1> in <expr2>** ist auch eine nicht direkt zu übersetzende Anweisung. Im korrelierenden G-Code erfolgt zuerst durch den *Alloc*

Befehl eine Bereitstellung von Heap-Knoten, in die die durch *letrec* gebundenen Variablen abgebildet werden[10].

Dann wird <expr1> ausgewertet. Es folgt für jedes Argument, das $< expr2 >$ von $< expr1 >$ übernimmt, ein *Push* Befehl. Die *Push* Befehle stellen die Argumente für $< expr2 >$ auf den Stack.

Der Befehl *Slide n* schiebt die Argumente nach der Auswertung von <expr2> wieder vom Stack. Das *n* entspricht immer der Anzahl der Argumente von <expr2>.

```
Alloc 1
<expr1>
Push0 (falls <expr2> genau ein Argument nimmt)
<expr2>
Slide 1
Update 0
Pop 0
Unwind
```

Wenn wir die STG-Anweisung **let <expr1> in <expr2>**, die im Aufbau der obigen *letrec* Anweisung entspricht, nach G-Code übersetzen, so ergibt sich der gleiche G-Code wie für die *letrec* Anweisung, nur daß bei der *let* Anweisung die G-Maschinen Instruktion *Alloc n* wegfällt.

Wir betrachten jetzt eine STG-Anweisung der Form **let binds in expr**, in der in *binds* mehrere Variablen gebunden werden.

Diese Möglichkeit, in *binds* mehrere Variablen zu binden, gibt es ausschließlich in der STG-Maschine. Für die G-Maschine müssen wir diese in verschachtelte *let*-Anweisungen übersetzen:

Die STG-Anweisung **let f=<expr 1>; g=<expr 2> in <expr 3>** wird dann zu der STG-Anweisung **let f=<expr1> in let g=<expr2> in <expr3>** und erst diese können wir in ihren korrelierenden G-Code übersetzen:

```
<expr1>
Push 0
<expr2>
Push 0
<expr3>
Slide1
Slide1
Update 0
Pop 0
Unwind
```

[10] Den *Alloc* Befehl brauchen wir nur bei *letrec*, weil nur hier evtl. folgende *let* oder *letrec* Anweisungen die schon definierten lokalen Variablen benutzen dürfen.

Dabei entfällt die *Push 0* Anweisung hinter $< expr1 >$, wenn der Ausdruck $< expr1 >$ keine Argumente nimmt und die *Push 0* Anweisung hinter $< expr2 >$, wenn der Ausdruck $< expr2 >$ keine Argumente nimmt.

Es lassen sich also alle STG-Code Konstrukte in äquivalente G-Code Konstrukte übersetzen.

Es gibt jedoch eine Einschränkung: Der STG-Code kennt freie Variablen, während die G-Maschine diese nicht zuläßt. Wenn wir ein STG-Programm mit freien Variablen in G-Code übersetzen wollen, so müssen wir das STG-Programm vor der Übersetzung in ein STG-Programm ohne freie Variablen transformieren.

Den Prozeß, alle freien Variablen aus Funktionsdefinitionen durch Transformation zu eliminieren, nennen wir *lambda lifting* (siehe [PJL91], Kapitel 6).

Wenn wir STG-Code *lambda-liften* wollen, so ist besonders einfach: Es entfällt nämlich der Schritt, herauszufinden, welche Variablen freie Variablen sind, da die freien Variablen im STG-Code ausgewiesen sind.

Im nächsten Abschnitt erstellen wir einen STG-To-G Übersetzer, der ein STG-Programm in den äquivalenten G-Code übersetzt.

5.3.2 Der STG-To-G Übersetzer

Der STG-To-G Übersetzer bekommt als Eingabe ein STG-Programm, daß ihm als abstrakter Datentyp *STGProgram*[11] vorliegt.

Dieses STG-Programm wird dann in eine Liste von Superkombinatoren übersetzt, die die folgende Form haben:

```
type GmCompiledSC = (Name, Int, GmCode)
```

Dabei gibt *Name* den Namen der G-Maschinen Funktion, der dem Namen der früheren STG-Funktion entspricht, an.

Int ist die Anzahl an Argumenten, die die Funktion braucht, um evaluiert werden zu können und *GmCode* ist der G-Maschinen Code der Funktion.

Um ein STG-Programm in diese Liste von kompilierten Superkombinatoren zu übersetzen, wird für jede STG-Funktion, die als Paar aus *(Var,LambdaForm)* dargestellt ist, die Funktion *compileSC* angewendet:

[11] Vgl. Implementierungsbeschreibung des STG-Interpreters.

```
compileSc :: (Var,LambdaForm) -> GmCompiledSC
compileSc (name,lambdaform)
      = (name,length(args), compileR stgexpr (zip args [0..]))
        where (vars,update,args,stgexpr)=lambdaform
```

Den Namen der STG-Funktion (*Var*) können wir direkt als Namen für den G-Maschinen Superkombinator übernehmen. Die Anzahl von Argumenten des G-Maschinen Superkombinators ergibt sich aus der Länge der *args* Liste, die in der Datenstruktur *LambdaForm*, die die STG-Funktion speichert, enthalten ist.

Um den G-Code für die STG-Funktion zu erzeugen, rufen wir die Funktion *compileR* auf, die als Argumente den STG-Ausdruck aus der *LambdaForm* (*stgexpr*), sowie die Argumente aus der *LambdaForm* bekommt.

Die Argumente der Lambdaform werden dabei als Liste von Paaren dargestellt. Dabei bilden die Argumente aus der Argumentliste der *LambdaForm* jeweils das erste Element eines Paares. Das zweite Element jedes Paares ist eine Zahl, die die Argumente von 0 an durchnumeriert. Dadurch bauen wir aus den Argumenten eine lokale Umgebung auf, die alle Argument-Namen eindeutig auf Zahlen abbildet. Wir nennen diese Abbildung *GmEnvironment*:

```
type GmEnvironment = Assoc Name Int
```

Die *compileR* Funktion, die einen STG-Ausdruck und die *GmEnvironment* als Argumente nimmt, hat dann den Typ:

```
type GmCompiler = STGExpr -> GmEnvironment -> GmCode
```

Da wir diesen Typ für mehrere Kompilierungsfunktionen brauchen, definieren wir ihn als Typvariable.

Die Funktion *compileR* ruft *compileE* auf, um den STG-Ausdruck nach G-Code zu übersetzen, und schreibt die drei G-Befehle *Update n, Pop n* und *Unwind* dazu.

Update n aktualisiert den Funktionsknoten mit dem neuen Wert; da die Funktion n Argumente hat, die von 0 bis $(n-1)$ durchnumeriert sind, steht an der Stelle n immer der Funktionsknoten.

Nachdem die Funktion mit dem Ergebnis aktualisiert wurde, wird das Ergebnis selbst nicht mehr gebraucht. Mit der Anweisung *Pop n* wird dieses vom Stack entfernt.

Unwind fährt schließlich mit der Evaluierung fort.

```
compileR :: GmCompiler
compileR e env = compileE e env ++
 [Update_ (length(env)), Pop (length(env)), Unwind]
```

Die Funktion *compileE* erzeugt nun den G-Code in Abhängigkeit des STG-Ausdrucks: Wenn der STG-Ausdruck aus einem Literal, also einem Integerwert n, besteht, wird dieser nach *Pushint n* übersetzt:

```
compileE (STGLiteral n) env = [Pushint n]
```

Die STG-Ausdrücke *Let* und *Letrec* werden übersetzt, indem die Funktion *compileLet* bzw. *compileLetrec* mit den Parametern *compileE*[12], *binds*[13], *expr*[14] und *env*[15], aufgerufen wird.

```
compileE (STGLet binds expr) env
 = compileLet compileE binds expr env
compileE (STGLetrec binds expr) env
 = compileLetrec compileE binds expr env
```

Ein STG-Ausdruck der Form *STGBiOp biop atoms*, wobei *biop* die *Built-In-Operation* ist und *atoms* die Atome sind, auf die diese angewendet werden soll, wird durch die Funktion *compileBiOp* behandelt:

```
compileE (STGBiOp biop atoms) env
 = compileBiOp (STGBiOp biop atoms) env
```

Der *case*-Ausdruck (*STGCase e alts*) wird so übersetzt, daß zunächst der Ausdruck *e* durch die Funktion *compileE* übersetzt wird, dem der G-Befehl *Casejump* mit den durch die Funktion *compileD* kompilierten Alternativen folgt:

[12]Der Kompiler, mit dem der Ausdruck des *let* oder *letrec* kompiliert werden soll.
[13]Die Variablen, die durch das *let* oder *letrec* gebunden werden.
[14]Der Ausdruck des *lets* oder *letrec*.
[15]Die Abbildung der Argumentnamen auf ihre Stackpositionen.

```
compileE (STGCase e alts) env
 = compileE e env ++ [Casejump (compileD alts env)]
```

Zur Kompilierung des *Cons* Befehls unterscheiden wir mehrere Möglichkeiten:

Zum einen, wenn *Cons* nur auf einen Konstruktor ohne Argumente angewendet wird, lautet die Übersetzung in G-Code *Pack c 0*; *c* ist dabei der *Tag* des Konstruktors, der dem *Tag* des STG-Konstruktors entspricht und 0 ist die Anzahl der Argumente.

Zum anderen, wenn der Konstruktor auf Literale oder Variablen angewendet wird. Dann werden Literale vor dem *Pack*-Befehl mit dem *Pushint* Befehl auf den Stack geschrieben. Variablen werden vor dem *Pack*-Befehl mit dem *Pushglobal* Befehl auf den Stack geschrieben, falls sie global sind und mit dem *Push n* Befehl, falls sie lokal sind. Dabei wird der Wert für *n* in der Umgebung *env*, die Variablen auf Stackpositionen abbildet, nachgeschlagen.

Die entsprechenden Kompiler-Schemata befinden sich im Anhang A. Die Kompiler-Schemata zur Kompilierung des *Cons* Befehls fangen dabei alle mit: compileE (STGCons ...) an.

Für alle sonstigen STG-Ausdrücke kompilieren wir den Ausdruck mit dem *compileC* Kompiler und geben danach den *Eval* Befehl, denn im Unterschied zu den mit *compileE* kompilierbaren Ausdrücken müssen die mit *compileC* kompilierten Ausdrücke noch durch *Eval* evaluiert werden:

```
compileE e env = compileC e env ++ [Eval]
```

Der compileC Kompiler betrachtet zunächst Applikationen:

Im einfachsten Fall, wenn eine Variable auf die leere Argumentliste angewendet wird, schreiben wir die Variable einfach mit *Pushglobal* auf den Stack, falls es eine globale Variable ist und mit *Push n*, falls es eine lokale Variable ist.

Den Wert *n* schlagen wir in der Abbildung der Namen auf die Stackpositionen nach.

Wird die Variable auf Argumente angewendet, so müssen wir mehrere Fälle berücksichtigen: Sind die Argumente Integerwerte, so schreiben wir diese mit *Pushint* auf den Stack. Sind es Variablen, dann schreiben wir diese mit *Pushglobal* oder *Push* auf den Stack.

Anschließend müssen wir für jedes Argument, auf das die Variable angewendet wird, einen *Mkap* Befehl geben.

Die entsprechenden Kompiler-Schemata befinden sich wieder im Anhang A unter dem Programmcode. Die Kompiler-Schemata für Applikationen beginnen alle mit:
compileC (STGApp ...).

Die *compileC* Übersetzungsschemata für Literale, *let*, *letrec*, Konstruktoren und eingebaute Operationen sind schon von *compileE* her bekannt. Der einzige Unterschied ist, daß beim Aufruf von *compileLet* bzw. *compileLetrec* als Kompiler *compileC* angegeben wird. Diese Übersetzungsschemata finden sich auch wieder in Anhang A. Die *compileC* Übersetzungschemata für Literale, *let*, *letrec*, Konstruktoren und eingebaute Operationen fangen mit:
compileC (STGLiteral ...) bzw.
compileC (STGLet ...) bzw.
compileC (STGLetrec ...) bzw.
compileC (STGCons ...) bzw.
compileC (STGBiOp ...) an.

Die Funktion *compileBiOp* übersetzt die STG-Ausdrücke, in denen eine eingebaute Operation auf zwei Atome angewendet wird, in G-Code.

Dabei unterscheidet sie 4 Fälle, je nachdem, ob die Atome zwei Literale, eine Variable und ein Literal oder zwei Variablen sind.

Zuerst werden die Atome auf den Stack geschrieben. Dies geschieht durch *Pushint* für Literale, und *Push* für Variablen. Hierbei ist zu beachten, daß in den Fällen, in denen das zweite Atom eine Variable ist, die mit *Push* auf den Stack geschrieben werden soll, die Stackposition, die die Umgebung für diese Variable zurückgibt nicht mehr stimmt, da mittlerweile etwas auf den Stack geschrieben wurde.

Wir müssen das Atom, das wir auf den Stack geschrieben haben, mitzählen. Der Befehl lautet jetzt: *Push* $(n+1)$.

Sind die beiden Atome auf den Stack geschrieben, so müssen wir nur noch den G-Maschinen Befehl geben, der die gewünschte eingebaute Operation mit den beiden Argumenten ausführt. Wir müssen nichts ausrechnen, da eine eingebaute Operation im STG-Ausdruck auch eine eingebaute Operation im G-Maschinen Code ist, d.h. die G-Maschine hat diesen Befehl eingebaut und berechnet das Ergebnis für uns.

```
compileBiOp (STGBiOp biop [(Literal i1),(Literal i2)]) env
 = [Pushint i2] ++ [Pushint i1] ++ bioptogminstr biop
compileBiOp (STGBiOp biop [(Variable v1),(Literal i2)]) env
 | (elem v1 (aDomain env))
    = [Pushint i2] ++ [Push (n1+1)] ++ bioptogminstr biop
 | otherwise = error "undeklared variable"
   where
```

```
   n1 = aLookup env v1 (error "cannot find variable")
compileBiOp (STGBiOp biop [(Literal i1),(Variable v2)]) env
 | (elem v2 (aDomain env))
    = [Push n2] ++ [Pushint i1] ++ bioptogminstr biop
 | otherwise = error "undeklared variable"
   where
   n2 = aLookup env v2 (error "cannot find variable")
compileBiOp (STGBiOp biop [(Variable v1),(Variable v2)]) env
 | ((elem v2 (aDomain env)) && (elem v1 (aDomain env)))
    = [Push n2] ++ [Push (n1+1)] ++ bioptogminstr biop
 | otherwise = error "undeklared variable"
   where
   n1 = aLookup env v1 (error "cannot find variable")
   n2 = aLookup env v2 (error "cannot find variable")
```

Die Übersetzung einer eingebauten Operation in der STG-Sprache in das G-Code Äquivalent übernimmt die Funktion *bioptogminstr*.

```
bioptogminstr :: Prim -> [Instruction]
-- wird von compileBiOp gebraucht
bioptogminstr biop
           | (biop == Plus)  = [Add_]
           | (biop == Minus) = [Sub_]
           | (biop == Mul)   = [Mul_]
           | (biop == Div)   = [Div_]
           | (biop == Mod)   = [Mod_]
           | (biop == Eq)    = [Eq_]
           | (biop == Lt)    = [Lt_]
           | (biop == Le)    = [Le_]
           | (biop == Gt)    = [Gt_]
           | (biop == Ge)    = [Ge_]
           | (biop == Sqrt)  = [Sqrt_]
           | (biop == Ne)    = [Ne_]
           | otherwise       = error "undefined operator"
```

Die Funktion *compileD* kompiliert die Alternativen einer *case*-Anweisung. Diese ruft *compileAlts* mit *compileE'* als Parameter auf:

```
compileD = compileAlts compileE'
```

Die Funktion *compileAlts* steht vor dem Problem, die Default-Alternativen aus dem STG-Code, die es im G-Code gar nicht gibt, in G-Code zu übersetzen.

Da eine Alternative im G-Code immer durch eine Integerzahl gekennzeichnet ist, wird die Default-Alternative zur Alternative, die durch -999 gekennzeichnet ist, übersetzt. Damit die G-Maschine die Default-Alternative korrekt ausführt, war eine kleine Modifikation der G-Maschine unumgänglich.

Diese Modifikation und die Erklärung, warum die Verwendung von -999 als Default-Kennzeichnung problemlos möglich ist, wird im nächsten Abschnitt beschrieben.

Die Funktion *compileAlts* gibt es für Konstruktor-Alternativen[16] und Integer-Alternativen[17]. Da die Version für Integer-Alternativen genauso wie die Version für Konstruktor-Alternativen arbeitet, beschreiben wir hier nur die Version für Konstruktor-Alternativen:

Im ersten Fall ist die Alternativen-Liste leer, und die Default- Alternative soll eine Variable binden (*DefaultVar*). Dann wird dies zu einer G-Code Alternative mit dem *Tag* -999 und dem Code, der durch *compileE expr env'* erzeugt wird und der Anweisung *Slide 1* übersetzt.

Die Anweisung *compileE expr env'* kompiliert den Ausdruck der Default-Alternative. Dabei ist *env'* die lokale Umgebung, in der berücksichtigt ist, daß jetzt eine neue, weitere Variable gebunden worden ist (die Default-Variable). Die Anweisung *Slide 1* wirft die Default-Variable nachher wieder vom Stack.

Der zweite Fall von *compileAlts*, wenn die Alternativen Liste leer ist und der Default-Fall keine Variable bindet, ist der einfachste Fall: jetzt wird nur der Default-Ausdruck ausgewertet.

Der dritte Fall von *compileAlts* tritt auf, wenn in der Alternativen-Liste noch Einträge vorhanden sind. Wir nehmen dann die erste Alternative, die aus dem Konstruktor, den Konstruktor-Argumenten und dem Ausdruck besteht. Die G-Code Alternative, die wir jetzt erzeugen, kennzeichen wir mit dem Konstruktor als *Tag*. Dann kompilieren wir den Ausdruck in der Umgebung, die wir bilden, indem wir die Konstruktor-Argumente auf den Stack schreiben.

Schließlich rufen wir wieder *compileAlts* auf, um die restlichen Alternativen zu kompilieren.

```
compileAlts comp (Algebraic ([],(DefaultVar var expr))) env
 = [(-999,compileE expr env'++[Slide 1])] --  f"ur default
   where env' = (argOffset 1 env) ++ [(var,0)]
compileAlts comp (Algebraic ([],(DefaultNoVar expr))) env
```

[16] Algebraic
[17] Primitive

```
 = [(-999,compileE expr env)] --  f"ur default
compileAlts comp (Algebraic (((constr,vars,stgexpr):aalts),
                  defaultalt)) env
 = [(constr,comp (length(vars)) stgexpr (zip vars [0..]
    ++ argOffset (length(vars)) env))]
    ++ compileAlts comp (Algebraic (aalts,defaultalt)) env

compileAlts comp (Primitive ([],(DefaultVar var expr))) env
 = [(-999,compileE expr env'++[Slide 1])] --  f"ur default
   where env' = (argOffset 1 env) ++ [(var,0)]
compileAlts comp (Primitive ([],(DefaultNoVar expr))) env
 = [(-999,compileE expr env)] --  f"ur default
compileAlts comp (Primitive (((lit,stgexpr):palts),
                  defaultalt)) env
 = [(lit,compileE stgexpr env)]
    ++compileAlts comp (Primitive (palts,defaultalt)) env
  where vars = []
```

Wenn in *compileAlts* als Argument *comp* auftauchte, dann war damit *compileE'* verbunden. Es wird aber fast immer nur als Argument durchgereicht, bis auf den dritten Fall von *compileAlts* für Konstruktor-Alternativen, wenn die Konstruktoren auf den Stack geschrieben werden. Dann wird nämlich, bevor der Ausdruck kompiliert wird, der G-Code *Split offset* eingefügt und nachher *Slide offset*.

Die Variable *offset* speichert die Anzahl der Konstruktor-Argumente. Mit dem Befehl *Split offset* werden die Konstruktor-Argumente auf den Stack geschrieben und mit *Slide offset* werden sie nach der Verwendung wieder vom Stack entfernt.

```
compileE' offset expr env
 = [Split offset] ++ compileE expr env ++ [Slide offset]
```

Es folgen mehrere Funktionen, die alle für die Kompilierung von *let* bzw. *letrec* Ausdrücken gebraucht werden.

Die Funktionen *compileLetrec* und *compileLet* werden direkt aufgerufen, um einen *let* oder *letrec* Ausdruck zu kompilieren.

Beide Funktionen sind identisch bis auf die Tatsache, daß *compileLetrec* erst einen *Alloc n* Befehl absetzt. Die Variable *n* entspricht dabei der Anzahl der Variablen, die gebunden werden sollen.

Der *Alloc* Befehl legt neue Heap-Knoten an. Damit wird erreicht, daß nachfolgende Bindungen von Variablen auf zuvor mit *letrec* gebundene Variablen zugreifen können.

Beide Funktionen rufen *compileLetrec'* bzw. *compileLet'* auf, um die Variablen-Bindung zu übersetzen, dann *comp expr env'*, um den Ausdruck des *let* oder *letrec* zu kompilieren und schließlich *Slide*, um die lokalen Variablen wieder vom Stack zu werfen.

Die Auswertung des Ausdrucks geschieht dabei in der Umgebung *env'*, die mit der Funktion *compileArgs* erzeugt wurde und die Umgebung so verändert, daß die durch *compileLetrec'* bzw. *compileLet'* neu auf den Stack gekommenen Variablen auch berücksichtigt werden.

```
compileLetrec :: GmCompiler -> Binds -> GmCompiler
compileLetrec comp binds expr env
 = [Alloc (length(binds))]
    ++ compileLetrec' binds env'
    ++ comp expr env' ++ [Slide (length(binds))]
   where env' = compileArgs binds env

compileLet :: GmCompiler -> Binds -> GmCompiler
compileLet comp binds expr env
  = compileLet' binds env
    ++ comp expr env'
    ++ [Slide (length(binds))]
    where env' = compileArgs binds env
```

Die Funktion *compileArgs* ordnet die Abbildung der Variablen auf Stackpositionen neu an: Die durch den *let* bzw. *letrec* Ausdruck neu hinzugekommenen Variablen haben die Stackpositionen $n-1$ bis 0. Die alten Werte auf dem Stack werden durch die Funktion *argOffset* in ihren Positionen um n nach oben verschoben.

```
compileArgs :: [(Name,LambdaForm)] -> GmEnvironment -> GmEnvironment
compileArgs binds env
     = zip (map fst binds) [n-1, n-2 .. 0] ++ argOffset n env
     where n = length(binds)

argOffset :: Int -> GmEnvironment -> GmEnvironment
```

```
argOffset n env = [(v, n+m) | (v,m) <- env]
```

Die Funktionen *compileLetrec'* und *compileLetrec* übersetzen die Variablenbindungen.

Wenn die Liste der zu bindenden Variablen leer ist, so brechen die Funktionen ab. Ansonsten wird eine Variablenbindung der Form *(name,lambdaform)* übersetzt.

Dabei wird mit *compileC* der STG-Ausdruck, der in der *LamdaForm* steht, kompiliert.

Den Namen, der in *name* gespeichert ist, beachten die Funktionen nicht. Die Zuweisung des Namens zu dem entsprechenden G-Code geschieht durch die Funktion *compileArgs*.

Nach Kompilierung der Variablenbindung nehmen sich die Funktionen die restlichen gebundenen Variablen vor, indem sie sich selbst wieder aufrufen.

Die Funktion *compileLetrec'* bleibt in der gleichen Umgebung, da die alte Umgebung wegen des *Alloc* Befehls nicht mehr aktualisiert werden muß. Die Funktion *compileLet'* fährt in einer aktualisierten Umgebung fort.

```
compileLetrec' :: [(Name,LambdaForm)] -> GmEnvironment -> GmCode
compileLetrec' [] env = []
compileLetrec' ((name,lambdaform):binds) env
 = compileC stgexpr env ++ compileLetrec' binds env
   where (vars,update,args,stgexpr)=lambdaform

compileLet' :: [(Name,LambdaForm)] -> GmEnvironment -> GmCode
compileLet' [] env = []
compileLet' ((name,lambdaform):binds) env
 = compileC stgexpr env ++ compileLet' binds (argOffset 1 env)
   where (vars,update,args,stgexpr)=lambdaform
```

5.3.3 Die modifizierte G-Maschine

Um die von dem STG-To-G Übersetzer erstellten G-Programme laufen zu lassen, müssen wir die G-Maschine leicht modifizieren: Im G-Code gibt es keine Default-Alternative. Doch wie soll nun eine im STG-Code vorhandene Default-Alternative von der G-Maschine behandelt werden?

Zunächst wird vom STG-To-G Übersetzer eine Default-Alternative in eine G-Code Alternative übersetzt, die den *Tag* -999 hat.

Die Funktion der G-Maschine, die wir verändern mußten, damit die G-Maschine diese Default-Alternative richtig behandelt, ist die angegebene Funktion *cjump2*:

```
cjump2 stacktop [] state = error "no match!"
cjump2 (NConstr t2 ss) ((t,c):rest) state
 | (t==t2)    = putCode (c++(getCode state)) state
 | (t== -999) = putCode (c++(getCode state)) state
 | otherwise = cjump2 (NConstr t2 ss) rest state
cjump2 (NNum i2) ((t,c):rest) state
 | (t==i2)    = putCode (c++(getCode state)) state
 | (t== -999) = putCode (c++(getCode state)) state
 | otherwise = cjump2 (NNum i2) rest state
cjump2 stacktop alts state
 = putCode ([Eval]++[Casejump alts]++getCode state) state
```

Die Funktion *cjump2* erhält entweder einen Konstruktor mit dem *Tag* t2 oder einen Integerwert mit dem *Tag* i2 als Selektor, sowie eine Liste der Alternativen, wobei die erste Alternative, bestehend aus dem *Tag* t und dem Code c betrachtet wird.

Zuerst wird abgefragt, ob der *Tag* der Alternative dem Selektor entspricht. Wenn ja, wird der Code der Alternative zur Ausführung gebracht. Diesen Fall gab es auch schon bei der alten G-Maschine.

Neu ist jetzt, daß wenn der *Tag* dem Selektor nicht entspricht, abgefragt wird, ob der *Tag* der Alternative -999 ist[18]. Wenn dies der Fall ist, wird der Code der Default-Alternative übernommen.

Auch der Sonderfall, daß zufällig eine Alternative auftaucht, die den *Tag* -999 hat, bereitet keine Probleme, da zuerst alle Nicht-Default-Alternativen betrachtet werden.

Somit kommt die Verzweigung für die Nicht-Default- Alternative mit dem *Tag* -999 vor der Default-Alternative mit dem *Tag* -999 und die G-Maschine wählt korrekt die Nicht-Default-Alternative aus.

Damit ist die G-Maschine auch in der Lage, die von der STG-Sprache übersetzten Default-Alternativen richtig zu verarbeiten.

Die Repräsentation der Default-Alternative als -999 ist allerdings nur eine interne Darstellung, da es den Leser eines nach G-Code übersetzen STG-Programms verwirren würde, wenn die -999 als Alternative in einer *case*-Anweisung auftaucht.

[18]Also: ob die Alternative die Default-Alternative ist.

Dafür, daß die Default-Alternative bei der Ausgabe des übersetzen G-Codes mit dem Schlüsselwort *default* gekennzeichnet wird, sorgt die modifizierte *showcase-jump* Funktion:

```
showcasejump [] = iStr""
showcasejump (-999,c)
 = iConcat[(iStr "default"),showInstructions c]
showcasejump ((a,c):ss)
 = iConcat[(iNum a),showInstructions c,showcasejump ss]
```

Die modifizierte *showcasejump* Funktion ersetzt die interne Darstellung der Default-Alternative als -999 durch das Schlüsselwort *default*.

Auch hier ist gewährleistet, daß eine evtl. vorhandene echte -999 als Alternative nicht in das Schlüsselwort *default* übersetzt wird, denn es gibt erstens am Ende jeder von STG-Code übersetzten Alternativenliste eine Default-Alternative und zweitens wird die -999 nur dann in das Schlüsselwort *default* übersetzt, wenn diese in der letzten Alternative der Liste vorkommt.

5.3.4 Der erweiterte STG-To-G Übersetzer

Bisher hat der STG-To-G Übersetzer die Update-Information, die jede STG-Funktion in Form des *Updatesflags* aufweist, ignoriert.

Wir erweitern jetzt den STG-To-G Übersetzer so, daß er im G-Code keinen Update vollzieht, wenn das *Update-Flag* nicht gesetzt ist und einen *Update in Place* vollzieht, wenn das *Update-Flag* gesetzt ist.

Dies hat zum einen den Vorteil, daß wir die STG-Maschine und die G-Maschine sinnvoll vergleichen können, indem wir das gleiche STG-Programm auf beiden Maschinen laufen lassen.

Zum anderen können wir untersuchen, ob die G-Maschine effizienter arbeitet, wenn sie Funktionen, deren *Update-Flag* nicht gesetzt ist[19], nicht aktualisiert.

Die Änderungen im STG-To-G Übersetzer beziehen sich nur auf zwei Funktionen: *compileSc* und *compileR*.

Die Funktion *compileSc* übergibt das Update-Flag an die Funktion *compileR*:

[19]Das *Update-Flag* einer Funktion ist nicht gesetzt, wenn von ihr feststeht, daß sie im Programmablauf nur einmal ausgewertet wird.

```
compileSc :: (Var,LambdaForm) -> GmCompiledSC
compileSc (name,lambdaform)
 = (name,length(args), compileR update stgexpr (zip args [0..]))
   where (vars,update,args,stgexpr)=lambdaform
```

Die Funktion *compileR* unterscheidet zwei Fälle: Wenn das Update-Flag gesetzt ist, gibt sie den Befehl *Updateip*, der die Funktionsrepräsentation im Heap mit dem Ergebnis überschreibt.

Wenn das Update-Flag nicht gesetzt ist, so findet keine Aktualisierung statt:

```
compileR :: Update -> GmCompiler
compileR Update e env
             = compileE e env ++
                [Updateip (length(env)), Pop (length(env)), Unwind]
compileR NoUpdate e env
         = compileE e env ++
           [Unwind]
```

5.3.5 STG-Programm mit gesetztem Update-Flag

Wir lassen das STG-Programm: $main = \{\} \backslash u \{\} \rightarrow +\{1,2\}$ vom erweiterten STG-To-G Übersetzer übersetzen und auswerten.

Der STG-To-G Übersetzer erzeugt folgenden G-Code:

```
Code for main
  Code:{Pushint 2
        Pushint 1
        Add
        Updateip 0
        Pop 0
        Unwind}
```

Wir sehen, daß im G-Code der neu definierte *Updateip* Befehl auftaucht, der ein *Update in Place* durchführt.

Bei der Ausführung des G-Programms beschränken wir uns auf die wesentlichen Schritte. In Schritt 7 hat die G-Maschine die Addition durchgeführt und das Ergebnis steht an der obersten Stackposition:

```
  7) Output://
       Stack:[1: Global main
              18: 3]
       Dump:[<{Print}, []>]
        Code:{Updateip 0
              Pop 0
              Unwind}
```

Im Schritt 8 wird der Heap-Knoten mit der Adresse 1, der die *main* Funktion repräsentiert, mit dem Ergebnis überschrieben:

```
  8) Output://
       Stack:[1: 3]
       Dump:[<{Print}, []>]
        Code:{Pop 0
              Unwind}
```

In Schritt 9 passiert nichts, da die *Pop 0* Anweisung 0 Elemente vom Stack entfernt.

In den folgenden Schritten wird das Ergebnis dann ausgegeben.

```
 10) Output://
       Stack:[1: 3]
       Dump:[]
        Code:{Print}


 11) Output:/3/
       Stack:[]
       Dump:[]
        Code:{}
```

5.3.6 STG-Programm mit nicht-gesetztem Update-Flag

Wir lassen das STG-Programm: $main = \{\} \backslash n \{\} \rightarrow +\{1, 2\}$ nach G-Code übersetzen und ausführen.

Das Programm wird zu folgendem G-Code übersetzt:

```
Code for main
  Code:{Pushint 2
        Pushint 1
        Add
        Unwind}
```

Der G-Code für das STG-Programm mit nicht-gesetztem Update-Flag ist, abgesehen vom dem Fehlen der *Updateip* und *Pop* Befehle, identisch zu dem G-Code für das STG-Programm mit gesetztem Update-Flag.

Wenn wir den G-Code durch die G-Maschine ausführen lassen, steht in Schritt 7 wieder das Ergebnis auf dem Stack.

In den folgenden Schritten wird das Ergebnis ausgegeben. Der Heap-Knoten, der die *main* Funktion enthält, wird aber nicht mit dem Ergebnis überschrieben.

```
7) Output://
     Stack:[1: Global main
            18: 3]
     Dump:[<{Print}, []>]
      Code:{Unwind}
```

Daraufhin wird das Ergebnis ausgegeben:

```
8) Output://
     Stack:[18: 3]
     Dump:[]
      Code:{Print}

9) Output:/3/
     Stack:[]
     Dump:[]
      Code:{}
```

5.3.7 Ergebnis des Vergleichs

Zur Ausführung des STG-Programms mit nicht-gesetztem Update-Flag hat die G-Maschine zwei Schritte weniger benötigt.

Wir haben uns den Schritt, den Funktionsknoten, der die *main* Funktion repräsentiert, mit dem Ergebnis zu überschreiben, sparen können, weil die *main* Funktion nur einmal im Programmablauf ausgewertet wird. Außerdem haben wir den Schritt gespart, die Ergebniswerte vom Stack zu entfernen[20].Die Ergebniswerte bleiben einfach als Endergebnis auf dem Stack stehen.

Als Resultat können wir damit festhalten, daß wir die G-Maschine durch Ausnutzung von Updateinformation optimieren können: Wenn für eine Funktion feststeht, daß diese nur einmal im Programmablauf ausgewertet wird, so setzen wir deren Update-Flag auf *nicht updaten* und sparen uns damit den Schritt, den Funktionsknoten mit dem Ergebnis zu überschreiben.

[20]Dies geschieht durch den Pop Befehl.

5.4 Vergleich: STG-Maschine und G-Maschine

5.4.1 Vergleich der Auswertung des STG-Fakultätsprogramms durch die STG und die G-Maschine

Mit dem STG-To-G Übersetzer, der auch die Updateinformation des STG-Programms berücksichtigt, sind wir jetzt in der Lage, das gleiche STG-Programm von der STG-Maschine und der G-Maschine auswerten zu lassen und somit beide Maschinen zu vergleichen.

Als STG-Programm mit dem wir diesen Vergleich ausführen, nehmen wir das Fakultätsprogramm:

```
Code for fak
fak = {} \n {x} -> case - {x,1} of
    0 -> 1;
    x1 -> case fak x1 of
    res -> * {x,res}

Code for main
main = {} \n {} -> fak 3
```

Die Ausführung durch die STG-Maschine lassen wir durch den STG-Maschinen Interpreter vornehmen. Die Ausführung befindet sich im Anhang B.

Die Übersetzung nach G-Code durch den STG-To-G Übersetzer und die Ausführung durch die G-Maschine unter Berücksichtigung der Updateinformation befindet sich ebenfalls in Anhang B.

Ablaufvergleich

Wir vergleichen den Ablauf der STG-Maschine mit der G-Maschine, die Updateinformation berücksichtigt und *Updates in Place* ausführt.

Den Stack der beiden Maschinen haben wir dabei in Abbildung 5.1 gegenübergestellt. In der Abbildung bedeutet z.B. *STG:1 G:2*, daß der Schritt 1 der STG-Maschine dem Schritt 2 der G-Maschine gegenübergestellt wird. Wenn ein Stack-Eintrag auf einen Heap-Knoten verweist, so haben wir den Inhalt des betreffenden Heap-Knotens gleich mitangegeben, z.B. *2: Global main*: auf dem Stack steht der Verweis auf den Heap-Knoten Nummer 2, der *Global main* enthält.

Um die Ablaufteile vorzustellen, die sich bei beiden Maschinen identifizieren lassen, liefern wir zu jedem Schritt der STG-Maschine den jeweils vergleichbaren Schritt der G-Maschine.

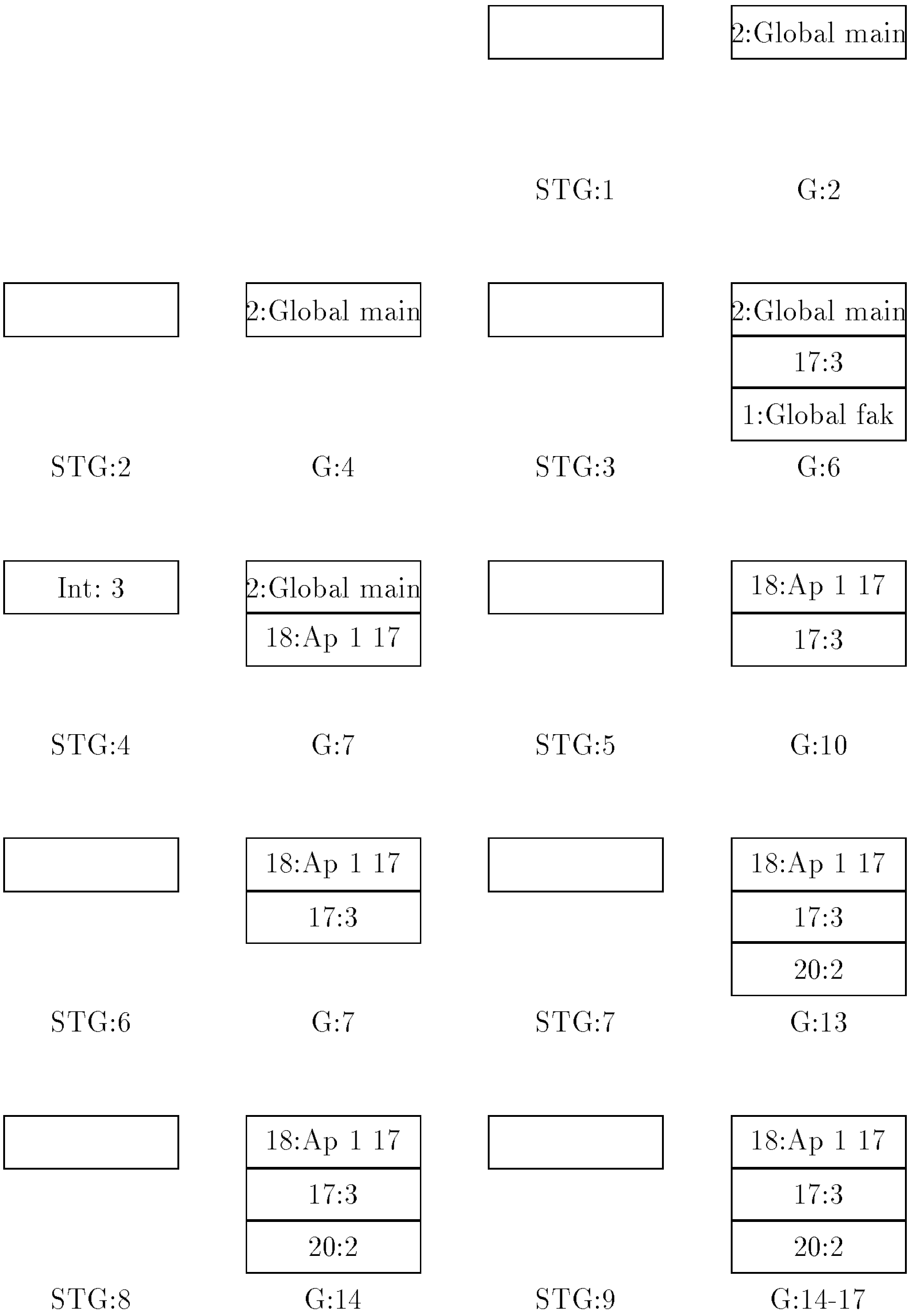

Abbildung 5.1: Gegenüberstellung des Stacks bei der STG- und G-Maschine

Zunächst geben beide Maschinen den Befehl, die *main* Funktion auszuwerten.

Dies geschieht bei der STG-Maschine in **Schritt 1** durch den Befehl *Eval main*. Bei der G-Maschine steht in **Schritt 2** *Global main* auf dem Stack und der Befehl *Eval* steht zur Auswertung in der Code-Komponente.

In **Schritt 2** gibt die STG-Maschine den Befehl *Enter 2*, der die rechte Seite der *main* Funktion, die im Heap an Adresse 2 steht, zur Auswertung bringt. Dies entspricht **Schritt 4** in der G-Maschine. Dort wird jetzt die rechte Seite der *main* Funktion durch die Befehle *Pushint 3, Pushglobal fak, Mkap* repräsentiert.

In **Schritt 3** gibt die STG-Maschine den Befehl *Eval fak 3*, der die Fakultätsfunktion auf die Integerzahl 3 anwendet. Dies entspricht **Schritt 6** der G-Maschine. Dort steht die Integerzahl 3 sowie die Fakultätsfunktion *Global fak* auf dem Stack. Der Befehl *Mkap* wendet die Fakultätsfunktion auf die Integerzahl 3 an.

In **Schritt 4** bringt die STG-Maschine die rechte Seite der Fakultätsfunktion mit dem Befehl *Enter 1* zur Auswertung. Das Argument, die Integerzahl 3, steht jetzt auf dem Argument-Stack. Dies entspricht **Schritt 7** der G-Maschine. Dort steht *Ap 1 17* auf dem Stack: die Applikation des Heap-Knotens 1 (die Fakultätsfunktion) auf den Heap-Knoten 17 (die Integerzahl 3). Der Befehl, der die Applikation ausführt, lautet *Eval*.

In **Schritt 5** gibt die STG-Maschine mit *Eval* den Befehl, die Fakultätsfunktion auszuwerten. Dabei wird in der lokalen Variablen x das Argument (die Integerzahl 3) gespeichert. Dies entspricht **Schritt 10** der G-Maschine. Dort steht das Argument auf dem Stack und die rechte Seite der Fakultätsfunktion befindet sich als G-Code in der Code-Komponente.

In **Schritt 6** wertet die STG-Maschine zunächst den Ausdruck $x - 1$ aus, den die Fakultätsfunktion für die Fallunterscheidung braucht. Dies geschieht durch den Befehl *Eval* $-\{x, 1\}$. Die lokale Variable x ist dabei an die Integerzahl 3 gebunden. In der G-Maschine wird zunächst auch dieser Ausdruck ausgewertet. Diese Auswertung findet in **Schritt 10 bis 12** statt. Dort werden die Befehle *Pushint 1, Push 1, Sub* nacheinander ausführt. Dabei wird erst die Zahl 1 auf den Stack geschrieben, dann die Zahl 3 daruntergesetzt und dann die Subtraktion ausgeführt.

In **Schritt 7** hat die STG-Maschine das Ergebnis der Subtraktion berechnet, das mit *Returnint 2* an die Fallunterscheidung weitergereicht wird. In der G-Maschine steht in **Schritt 13** das Ergebnis auf dem Stack. In der Code-Komponente befindet sich, eingeleitet durch *Casejump*, der Code, der die Fallunterscheidung ausführt.

In **Schritt 8** hat die STG-Maschine durch Vergleiche festgestellt, welcher Fall der Fallunterscheidung zutrifft: der Default-Fall. Jetzt bringt sie den STG-Code für den Default-Fall mit *Eval* zur Auswertung. Dabei speichert sie das Ergebnis der Subtraktion als lokale Variable *x1* und das Argument als lokale Variable *x*. Dies entspricht **Schritt 14** der G-Maschine. Diese hat den G-Code für den Default-Fall bereits in die Code-Komponente geschrieben. Das Ergebnis der Subtraktion und das Argument stehen noch auf dem Stack.

In **Schritt 9** wertet die STG-Maschine den Code für den Default-Fall aus. Dazu soll die Fakultätsfunktion auf das Ergebnis der Subtraktion angewendet werden. Dies geschieht durch den Befehl *Eval fak x1*, wobei die lokale Variable *x1* an das Ergebnis der Subtraktion gebunden ist. Die G-Maschine wertet den Code für den Default-Fall in **Schritt 14 bis 17** aus. Sie schreibt erst mit *Push 0* das Ergebnis der Subtraktion auf den Stack, dann mit *Pushglobal fak* den Heap-Knoten, der die Fakultätsfunktion enthält und schließlich gibt sie mit *Mkap* den Befehl, die Fakultätsfunktion auf das Ergebnis der Subtraktion anzuwenden.

Wie wir sehen, wenden beide Maschinen wieder die Fakultätsfunktion auf ein Argument an. Die Auswertungsmechanismen wiederholen sich jetzt bei beiden Maschinen wieder und wir geben sie deshalb nicht weiter an.

Anhand der gegenübergestellten Ablaufschemata können wir feststellen, daß die STG-Maschine und die G-Maschine die gleichen Einzelschritte unternehmen, um ein funktionales Programm auszuwerten.

Die Maschinen unterscheiden sich dabei nur in Details:

Die G-Maschine schreibt zunächst den Funktionsnamen der Funktion, die ausgewertet werden soll, auf den Stack und gibt dann den *Eval* Befehl, während die STG-Maschine den Funktionsnamen dem *Eval* Befehl als Argument mitgibt.

Die G-Maschine braucht deshalb mehr Schritte als die STG-Maschine, weil oft ein STG-Befehl eine Aktion in einem Schritt vollzieht, für den die G-Maschine bis zu drei Schritte benötigt. Der STG-Befehl führt aber trotzdem genau die gleiche Aktion aus, wie die drei G-Befehle.

Wir können den STG-Befehl in diesem Zusammenhang als eine Art Makro-Befehl, wie in der Maschinenspracheprogrammierung, betrachten.

Vergleich der benötigten Ressourcen

Die G-Maschine benötigt für die Ausführung des Programms 20 Heap-Knoten, die STG-Maschine nur 2.

Dies liegt daran, daß die STG-Maschine intensiver mit der Code-Komponente arbeitet. Die STG-Maschine hält z.B. Zwischenergebnisse, die die G-Maschine im Heap ablegt, in der Code-Komponente verfügbar.

Da die STG-Maschine deshalb mehr Speicher für die Code-Komponente braucht, können wir nicht sagen, daß die G-Maschine insgesamt mehr Speicherplatz braucht, nur weil sie mehr Heap-Knoten verwendet.

5.4.2 Vergleich: STG-Befehle und G-Befehle

Im letzten Abschnitt hat sich gezeigt, daß oft mehrere G-Code Befehle die gleiche Wirkung haben, wie ein STG-Maschinenbefehl.

Die STG-Maschine kennt die Instruktionen **Eval** *expr*, **Enter** *int*, **ReturnInt** und **ReturnCon**.

Die STG-Maschinenbefehle **Enter**, **ReturnInt** und **ReturnCon** bestimmen das globale Verhalten der STG-Maschine und werden im nächsten Abschnitt besprochen.

Wir stellen hier dem STG-Maschinenbefehl **Eval** *expr* für alle möglichen STG-Ausdrücke, *expr*, die äquivalente Sequenz von G-Code Instruktionen gegenüber und vergleichen, wie die operationalen Semantiken der Maschinen die Befehle verarbeiten.

- Der STG-Maschinenbefehl **Eval** *literal* wertet den STG-Ausdruck, der aus einem Literal besteht, aus. Wenn die STG-Maschine auf **Eval** *literal* trifft, schreibt sie den Wert *literal* in das Return-Register und gibt dann den *ReturnInt* Befehl (siehe *ReturnInt*).

  Die äquivalente G-Code Sequenz schreibt das Literal mit *Pushint* auf den Stack und gibt mit *Unwind* den Befehl, weiterzuarbeiten:

  Pushint *literal*
  Update 0
  Pop 0
  Unwind

- Der STG-Maschinenbefehl **Eval** $<$ *constr* $>$ { *literal1*, *literal2* } führt zur Auswertung des Konstruktors *constr* mit den Argumenten *literal1* und *literal2*. Die STG-Maschine schreibt den Konstruktor und dessen Argumente in die Return-Register und fährt mit dem *ReturnCon*-Befehl fort (siehe *ReturnCon*).

  Der G-Code schreibt diese Argumente mit *Pushint* auf den Stack und gibt dann den *Pack constr arity* Befehl, der *arity* Argumente an den Konstruktor *constr* bindet: [21]

  Pushint *literal2*
  Pushint *literal1*
  Pack *constr* 2
  Update 0

[21] Wir sehen, daß im G-Code schon das globale Verhalten der G-Maschine vorgezeichnet ist. Bei der STG-Maschine können wir STG-Befehle und globales Verhalten trennen, bei der G-Maschine geht das nicht.

Pop 0
Unwind

- Der STG-Maschinenbefehl **Eval** $\oplus$ { *literal1* , *literal2* } führt die eingebaute Operation $\oplus$ auf die Argumente *literal1* und *literal2* aus. Die STG-Maschine schreibt das Ergebnis in das Return-Register und fährt mit *ReturnInt* fort (siehe *ReturnInt*).

  Im G-Code werden die Argumente mit der *Pushint* Anweisung auf den Stack geschrieben. Dann wird der Befehl *biop* gegeben, der die eingebaute Operation auf die beiden obersten Stackelemente anwendet. Wir können jede eingebaute Funktion $\oplus$ der STG-Maschine eindeutig auf eine eingebaute Operation der G-Maschine abbilden[22]. Der G-Code lautet:

  Pushint *literal1*
  Pushint *literal2*
  biop
  Update 0
  Pop 0
  Unwind

- Der STG-Maschinenbefehl **Eval** *var* { *literal* }, beschreibt die Anwendung der Funktion *var* auf das Literal *literal*. Die STG-Maschine schreibt den Wert *literal* auf den Argument-Stack[23] und ruft die Funktion *var* durch den STG-Befehl *Enter a* auf, wobei *a* die Adresse des Heap-Knotens ist, an dem die Funktion *var* gespeichert ist.

  Im G-Code wird die Anwendung der Funktion *var* auf das Literal *literal* durch den *Pushint*-Befehl (der das Argument *literal* auf den Stack schreibt), den *Pushglobal*-Befehl (der die Funktion *var* auf den Stack schreibt), den *Mkap*-Befehl (der einen neuen Stack-Eintrag bildet, der die Anwendung der Funktion auf das Argument darstellt), sowie den *Eval*-Befehl (der die Anwendung auswertet), ausgedrückt:

  Pushint *literal*
  Pushglobal *var*
  Mkap
  Eval
  Update 0
  Pop 0
  Unwind

[22] Z.B. bilden wir die STG-Maschinenoperation + auf den G-Code-Befehl **ADD** ab.
[23] Den B-Stack, weil *literal* unboxed ist.

- Der STG-Maschinenbefehl **Eval letrec** *var* = $\{\}\backslash u\{\} \rightarrow$ *expr1 in expr2* stellt eine lokale Variable *var* mit dem Wert *expr1* zur Verfügung. Im dem STG-Ausdruck *expr2* kann diese lokale Variable benutzt werden. Dazu wird *Eval expr2* aufgerufen und die lokale Umgebung, die die Variable *var* an *expr1* bindet, wird mitgegeben.

  Im G-Code wird durch die Anweisung *Alloc 1* ein Heap-Knoten zur Aufnahme der lokalen Variablen bereitgestellt. Dann wird der Ausdruck *expr1* durch den entsprechenen G-Code ausgewertet. Durch die Anweisung *Push 0* wird das Ergebnis der Auswertung oben auf den Stack geschrieben. Der G-Code für den STG-Ausdruck *expr2*, der jetzt ausgewertet wird, greift auf die lokale Variable über die erste Stackposition zu. Schließlich wird mit *Slide 1* die lokale Variable vom Stack entfernt:

  Alloc 1
  G-Code für expr1
  Push 0
  G-Code für expr2
  Slide 1
  Update 0
  Pop 0
  Unwind

  Wenn wir im obigen STG-Ausdruck *let* anstatt *letrec* verwenden, so ergibt sich der gleiche G-Code, bis auf den Unterschied, daß die *Alloc 1* Anweisung fehlt. Der *Alloc 1* Befehl dient dazu, einen Heap-Knoten bereitzustellen, in den die durch *letrec* gebundene Variable geschrieben wird. Dadurch kann in weiteren *letrec* oder *let* Umgebungen auf diese Variable zugegriffen werden.

- Bei der Verzweigung mit *case* wird jede *case*-Alternative im STG-Code zu einem *Casejump* im G-Code übersetzt.

  Die *Default-Alternative* im STG-Code wird zu *Casejump -999* übersetzt:

  Eval case *expr1* **of** *tag*$\{\} \rightarrow$ *expr2*
  default $\rightarrow$ *expr3 endcase*

  (*G-Code für expr1*)
  Casejump *tag1*: (*G-Code für expr2*)
  Casejump -999: (*G-Code für expr3*)
  Update 0
  Pop 0
  Unwind

Beide Maschinen werten erst den Ausdruck hinter der *case*- Anweisung aus und wählen dann die passende Alternative aus.

5.4.3 Vergleich: globales Verhalten STG- und G-Maschine

Wir vergleichen das globale Verhalten der STG-Maschine und der G-Maschine bezüglich Startzustand und Übergabemechanismen für Argumente und Rückgabewerte (Ergebnisse).

- **Startzustand**

  Sowohl die STG-Maschine als auch die G-Maschine beginnen die Ausführung des Programms durch die Auswertung der *main*-Funktion.

  Die **STG-Maschine** gibt zuerst den Befehl *Eval main*, der dann zu dem Befehl *Enter a* führt, wobei *a* die Heap-Adresse ist, unter der der Code für die *main* Funktion zu finden ist. Der Befehl *Enter a* schreibt den Code der *main* Funktion in die Code-Komponente zur weiteren Auswertung.

  Die **G-Maschine** startet mit der *main* Funktion (Global main) auf dem Stack und gibt den *Eval* Befehl. Das Resultat ist, daß der Code für die *main* Funktion zur Auswertung auf dem Stack steht.

  Obwohl beide Maschinen anscheinend eine andere Aktion durchführen, bewirken sie doch mit diesen Aktionen dasselbe: Der Code der *main* Funktion wird zur Auswertung gebracht.

  Es zeichnet sich hier schon ab, daß die Auswertung bei der STG-Maschine hauptsächlich in der Code-Komponente stattfindet (der Code der *main*-Funktion wird zur Auswertung in die Code-Komponente geschrieben) und bei der G-Maschine überwiegend im Stack (der Code der *main*-Funktion wird zur Auswertung auf den Stack geschrieben).

- **Argumente**

  Übernimmt eine **STG-Funktion** *f* ein Argument von einer anderen STG-Funktion *g*, so wird die Funktion *f* aufgerufen mit: **Eval f** *arg*. Das Argument, *arg*, wird bei der Ausführung dieses *Eval* Befehls auf den Argument-Stack (A-Stack für Adressen, B-Stack für Literale) geschrieben und der Befehl *Enter a* wird gegeben, wobei *a* die Adresse des Heap-Knotens ist, der den Code von *f* speichert. Durch *Enter a* wird der Code von *f* ausgeführt, wobei das Argument vom Argument-Stack verwendet wird.

  Wenn die **G-Maschine** den Code für eine Funktion ausführt, die ein Argument verwendet, so ist gewährleistet, daß das Argument vor der Ausführung der Funktion an der obersten Stackposition steht. Mit *Push 0* wird sichergestellt, daß das Argument an der Stackspitze steht. Nach der Auswertung wird die Funktion *f*, die an Stackposition 1 steht, durch *Update 1* mit dem Ergebniswert der Funktion aktualisiert und das Argument wird mit *Pop 1* wieder vom Stack entfernt:

Push 0
G-Code der Funktion f
Update 1
Pop 1
Unwind

Auch wenn beide Maschinen andere Aktionen durchführen, so verwenden doch beide die gleiche Methode, um Funktionen, die Argumente erwarten, auszuwerten: sie stellen das Argument erst bereit und wenden dann die Funktion darauf an (*tail-call*).

- **Rückgabewerte**

  Liefert eine **STG-Funktion** einen *Rückgabewert*, so fährt die STG-Maschine mit *Returnint y* fort, wenn der Rückgabewert y ein Literal ist, oder mit *ReturnCon y*, wenn der Rückgabewert y ein Konstruktor ist. Die Befehle *ReturnInt* bzw. *ReturnCon* wählen die passende Alternative der *case*-Anweisung aus, falls der Rückgabewert das Ergebnis des Ausdrucks einer *case*-Anweisung war oder geben den Wert als Ergebnis aus, falls nicht.

  Bei der **G-Maschine** bleibt der Rückgabewert einfach auf dem Stack stehen, wo er erzeugt wurde und wird dann automatisch zum Argument für eine andere Funktion, zum Selektor, der eine Alternative der *case*-Anweisung auswählt, oder zum Ergebnis des Programms.

Auch bei der Behandlung von Argumenten und Rückgabewerten durch die STG-Maschine und die G-Maschine wird deutlich, daß die G-Maschine überwiegend mit dem Stack arbeitet, während die STG-Maschine den Stack (A- und B-Stack) nur kurzzeitig als Zwischenspeicher benutzt.

Bei der G-Maschine findet die eigentliche Programmauswertung auf dem Stack statt, bei der STG-Maschine in der Code-Komponente.

5.4.4 Vergleich: Implementierungsmöglichkeiten STG-Maschine und G-Maschine

Die STG-Maschine kann durch explizite Registermanipulationsbefehle optimiert werden, die G-Maschine nicht ohne weiteres:

Die G-Maschine führt *Graph-Rewriting* aus, d.h. der Graph der Funktion wird auf dem Stack aufgebaut und dann ausgewertet. (Deswegen stellen wir fest, daß die G-Maschine sich wesentlich stärker am Stack orientiert, als die STG-Maschine.)

Wollten wir die G-Maschine durch Registermanipulationsbefehle erweitern, müssten wir den kompletten Stack durch Register nachbauen. Ein Stack aus

Registern bringt jedoch weder bei der STG-Maschine, noch bei der G-Maschine Vorteile.

Um die G-Maschine durch Registermanipulationsbefehle optimieren zu können, müssten wir sie von der starken Stackbenutzung lösen, indem wir die Auswertung auf die Code-Komponente verlagern (z.B. Übergabe von Argumenten in der Code-Komponente).

Das Ergebnis einer solchen Umgestaltung der G-Maschine wäre aber die STG-Maschine. In diesem Sinne können wir die STG-Maschine als optimierte G-Maschine betrachten.

5.5 Resultate

- Alle Befehle der STG-Maschine können eindeutig auf eine Sequenz von G-Code abgebildet werden. Da oft ein einziger STG-Maschinenbefehl bis zu drei G-Maschinenbefehlen entspricht, können wir die STG-Instruktionen als eine Art Makrosprache für G-Maschinen Instruktionen ansehen.

- Im globalen Verhalten beider Maschinen lassen sich Parallelen feststellen. Es wird jedoch insbesondere bei der Betrachtung der Übergabe von Argumenten und Rückgabewerten deutlich, daß sich die Auswertung bei der G-Maschine fast ausschließlich auf dem Stack abspielt, während die Auswertung bei der STG-Maschine hauptsächlich in der Code-Komponente stattfindet.

- Da sich die G-Maschine wesentlich stärker an der Stackarchitektur orientiert, kann sie nicht ohne weiteres durch die Verwendung von expliziten Registermanipulationsbefehlen optimiert werden. Die Transformation der G-Maschine zu einer Maschine, die sich durch Registermanipulationsbefehle optimieren läßt, resultiert in der STG-Maschine. Deswegen können wir die STG-Maschine als optimierte G-Maschine betrachten.

Als Fazit können wir festhalten, daß der grundlegende Unterschied zwischen der STG-Maschine und der G-Maschine aus der stärkeren Stackorientierung der G-Maschine besteht. Ansonsten sind die Maschinen nicht so verschieden, wie sie auf den ersten Blick erscheinen, da sie die gleichen Konzepte zur Auswertung eines funktionalen Programms anwenden.

Kapitel 6

weitere Maschinen

6.1 Die ABC-Maschine

6.1.1 Einordnung der ABC-Maschine

Die abstrakte ABC-Maschine dient, wie die STG- und die G-Maschine, zur Auswertung von funktionalen Programmen.

Dabei wird ein funktionales Programm in einer funktionalen Sprache wie Miranda[Tur85] in die Zwischensprache Clean[1] übersetzt. Das Clean-Programm wird zur Ausführung in ABC-Code übersetzt, der entweder interpretativ von einem ABC-Maschinen Interpreter ausgeführt werden kann oder nach Maschinensprache übersetzt werden kann (siehe Abbildung 6.1).

Die *Template Instantiation Machine[PJL91]* wertet funktionale Programme durch *Graph Rewriting* aus. Da die Graphen in abstrakten Datenstrukturen gespeichert werden, führen einfache Kalkulationen, die ein heutiger Rechner sehr effizient ausführen kann, zu speicher- und rechenzeitaufwendigen Vorgängen.

Die ABC-Maschine hat dies berücksichtigt und besteht deshalb aus zwei Teilen: Einem Teil, der funktionale Programme durch *Graph Rewriting* auswertet und einem Teil, der einfache Berechnungen unter Benutzung von Stacks effizient ausführen kann.

6.1.2 Komponenten der ABC-Maschine

Die Komponenten der ABC-Maschine lassen sich in Komponenten, die zu dem *Graph Rewriting* ausführenden Teil gehören, solche, die zu dem Teil gehören, der

[1] Vergleichbar mit der Core-Language.

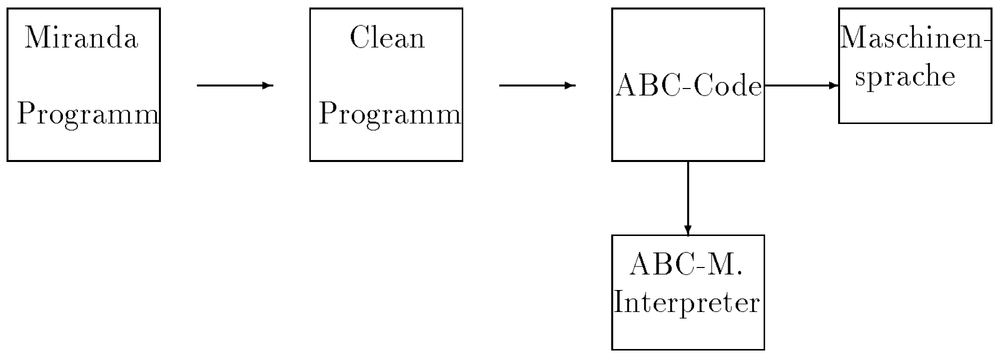

Abbildung 6.1: Einordnung der ABC-Maschine

Berechnungen mit Stacks ausführt und solche, die von beiden Teilen benutzt werden, unterteilen. Zu dem Teil der ABC-Maschine, der *Graph Rewriting* ausführt, gehören folgende Komponenten:

- Der *Graph-Speicher*, der das als Graph gespeicherte funktionale Programm enthält, das ausgewertet werden soll.
- Der *A-Stack* (Argument-Stack). Dieser verweist auf Knoten im Graphen-Speicher.
- Der *Bezeichner-Speicher*: Die ABC-Maschine rechnet intern nicht mit symbolischen Bezeichnern wie z.B. *Cons*, sondern mit numerischen Bezeichnern.

  Zur Ausgabe müssen wir wissen, welcher numerische Bezeichner auf welches Symbol abgebildet wird, da wir die Symbole ausgeben. Diese Abbildung speichert der Bezeichner-Speicher.

Zu dem Teil der ABC-Maschine, der einfache Kalkulationen mit Stacks effizient ausführen kann, zählen folgende Komponenten:

- Der *B-Stack* (basic-value-Stack): Der B-Stack dient der Speicherung von Integerwerten (basic values[2]).

Die weiteren Komponenten der ABC-Maschine werden von beiden Teilen benutzt:

- Der *Programm-Speicher*, der das auszuwertende ABC-Programm enthält.

[2] Die *basic values* der ABC-Maschine entsprechen den *unboxed values* der STG-Maschine.

- Der *Programmzähler*, der anzeigt, welcher Befehl im Programm-Speicher ausgeführt werden soll.
- Der *C-Stack* (Control-Stack). Der C-Stack speichert Rücksprungadressen.
- Der *Ein- und Ausgabekanal*, um die Ergebnisse auszugeben.

6.1.3 Arbeitsweise der ABC-Maschine

Ein *Clean* Programm wird in eine Folge von ABC-Instruktionen, das ABC-Programm übersetzt. Dieses ABC-Programm wird im *Programm-Speicher* gehalten.

Der Graph, der im Graphen-Speicher gehalten wird, stellt die Eingabe dar, die das funktionale Programm bearbeiten soll[3]. Die ABC-Instruktionen geben an, wie der Graph zur Normalform reduziert werden kann.

6.1.4 Basic values contra unboxed values

Ein wesentlicher Unterschied zwischen der STG-Maschine und der ABC-Maschine ist die Verwendung von Integerwerten, also den *unboxed values* bei der STG-Maschine bzw. *basic values* bei der ABC-Maschine.

In der ABC-Maschine hat jede Funktion zwei *Entry Points*: Der erste *Entry Point* wertet einen Ausdruck aus, bis dieser als *basic value* vorliegt und der zweite geht davon aus, daß der Ausdruck schon ausgewertet, also ein *basic value* ist.

In der STG-Maschine wurde ein anderer Weg genommen: Wir benötigen hier zwei Ausprägungen einer Funktion: Einen *Wrapper* (z.B. *fak_wrapper*), der einen Ausdruck auswertet und zu einem *unboxed value* transformiert und einen *Worker* (z.B. *fak_worker*), der mit den *unboxed values* arbeitet. Die Verwendung von zwei *Entry Points* für eine Funktion ist in der STG-Maschine nicht vorgesehen.

6.2 Die Three-Instruction Machine

6.2.1 Einordnung der Three-Instruction Machine

Die *Three-Instruction Machine* ist eine abstrakte Maschine, die wie die G-Maschine ein funktionales Programm in Core-Language auswertet. Sie wurde von Fairbairn und Wray[FW87] entwickelt.

[3] Die Eingabe, die durch ein funktionales Programm bearbeitet werden soll, ist in vielen Fällen und auch hier Teil des funktionalen Programms. Beispiel: main = fak 3; fak x = ...

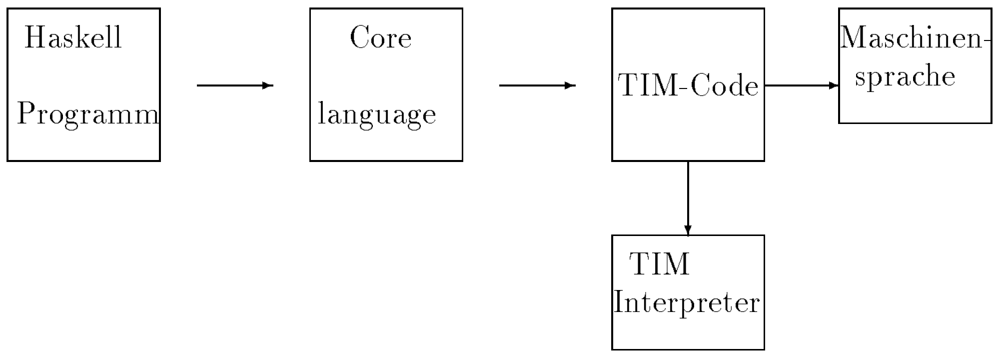

Abbildung 6.2: Einordung der Three-Instruction Machine

Wie bei der G-Maschine wird ein funktionales Programm, das z.B. in Haskell geschrieben sein kann, in die Zwischensprache *Core-Language* übersetzt. Das Core-Language Programm wird dann in den TIM-Code[4] übersetzt. Dieser kann von einem *Three-Instruction Machine* Interpreter ausgeführt werden oder weiter nach Maschinensprache übersetzt werden (siehe Abbildung 6.2).

Ihren Namen hat die *Three-Instruction Machine* daher, daß der TIM-Code im wesentlichen aus drei Befehlen besteht.

Die *Three-Instruction Machine* unterscheidet sich von der bekannten G-Maschine grundlegend in drei Punkten:

- *flache Applikationen.*

  Die *Three-Instruction Machine* transformiert das Core-Language Programm so, daß nur noch flache Applikationen auftreten. *Flache Applikationen* sind in dieser Arbeit schon im Zusammenhang mit der STG-Maschine vorgestellt worden[5]. Die *Three-Instruction Machine* definiert flache Applikationen etwas anders: Sie erlaubt die Funktionsanwendung nur auf maximal zwei Argumente (z.B. f x y).

  Die *Three-Instruction Machine* ersetzt in dem Core-Language Programm alle Funktionen, die mehr als zwei Argumente verwenden, durch Einführung von Hilfsfunktionen in Funktionen, die genau zwei Argumente verwenden.

  Dies hat gegenüber der G-Maschine den Vorteil, daß der *Spine*[6] nicht übermäßig groß werden kann.

[4] Three-Instruction Machine Code

[5] Bei der STG-Maschine sind Funktionsanwendungen nur auf Variablen oder Konstanten erlaubt.

[6] Die G-Maschine baut eine zu reduzierende Funktion erst als Graph auf. Diesen Graphen nennen wir *Spine*.

- *Gemeinsame Benutzung von Argumenten.*

  Wenn zwei Funktionen die gleichen Argumente benutzen, dann werden die gleichen Argumente zweimal im Heap angelegt. Um dies zu vermeiden, bildet die *Three-Instruction Machine* Tupel aus den gemeinsam benutzten Argumenten. Die Funktionen, die die gleichen Argumente benutzen, arbeiten dann mit dem gleichen Tupel.

- *Die Three-Instruction Machine verwendet keinen Spine.*

  Wenn wir für alle Funktionen nur noch Tupel als Argumente zulassen, dann sehen alle Funktionsanwendungen gleich aus: *Funktionsname tupel.* Wir können es uns jetzt ersparen, diese Funktionsanwendungen erst als Graph aufzubauen. Deswegen hat die *Three-Instruction Machine* keinen Spine.

6.2.2 Arbeitsweise der Three-Instruction Machine

Wir wollen das Core-Language Programm: *compose2 f g x = f (g x x)* durch die *Three-Instruction Machine* ausführen lassen.

Dazu werden im ersten Schritt *flache Applikationen* eingeführt. Die transformierte Form des Core-Language Programms lautet nun:

compose2 f g x = f (c1 g x),
c1 g x = g x x.

Das transformierte Core-Language Programm wird jetzt in TIM-Code übersetzt. Der TIM-Code für *compose2* lautet:

Take 3
Push (Label "c1")
Enter (Arg 1).

Der TIM-Code für *c1* lautet:

Push (Arg 3)
Push (Arg 3)
Enter (Arg 2).

Wir sehen, daß es sich bei den drei Befehlen der *Three- Instruction Machine* um die Befehle *Push*, *Take* und *Enter* handelt.

Schließlich wird der TIM-Code durch die *Three-Instruction Machine* ausgeführt. Dazu werden zuerst die Argumente der *compose2* Funktion auf den Stack geschrieben:

x
g
f

Jetzt wird der Code der *compose2* Funktion ausgeführt. Der erste Befehl, *Take 3* nimmt die drei Elemente vom Stack und bildet ein Tupel daraus. Der Stack ist nun leer und ein spezieller Zeiger (der *Frame Pointer*) zeigt auf das Tupel mit den drei Argumenten.

Als nächster Befehl wird *Push (Label "c1")* ausgeführt. Dieser schreibt die Befehle der *c1* Funktion in den Code-Speicher und bringt diese Funktion somit zur Ausführung.

Es werden jetzt die Befehle *Push (Arg 3), Push (Arg 3)* und *Enter (Arg 2)* ausgeführt. Die Angaben *Arg x* beziehen sich dabei auf die im Tupel gespeicherten Argumente. Dabei schreibt *Push (Arg 3)* das dritte im Tupel gespeicherte Argument *(x)* auf den Stack. Mit dem erneuten *Push (Arg 3)* wird nochmal *x* auf den Stack geschrieben. Schließlich wird mit *Enter (Arg 2)* das zweite Element im Tupel *(g)* darauf angewendet.

Wir haben die Hilfsfunktion *c1* ausgeführt. Auf dem Stack steht jetzt das Ergebnis der Anwendung von *g (x x)*:

g (x x)

Als letzter Befehl bleibt *Enter (Arg 1)* der Funktion *compose2*. Dieser wendet das erste Argument des Tupels *(f)* auf den Stackinhalt an.

Auf dem Stack steht jetzt das Ergebnis der Anwendung von *f* auf *g (x x)*:

f g (x x)

6.3 Resultate

Die *ABC-Maschine* und die *Three-Instruction Machine* unterscheiden sich nicht sehr von der bekannten *STG-Maschine* und der *G-Maschine*:

Die *ABC-Maschine* arbeitet, wie die *STG-Maschine*, mit Stacks, die die Argumente aufnehmen und führt Berechnungen ohne das ineffiziente *Graph-Rewriting* aus.

Die *Three-Instruction Machine*, die auf den ersten Blick von der G-Maschine völlig verschieden erscheint, ist im wesentlichen eine G-Maschine, die flache Applikationen verwendet, die Argumente in Tupeln speichert und keinen Spine besitzt.

Kapitel 7

Zusammenfassung

Um die Grundlagen für den Vergleich, die Analyse und die Optimierung der abstrakten Maschinen zur Auswertung nicht strikter funktionaler Sprachen zu schaffen, stellen wir in **Kapitel 2** funktionale Sprachen und deren Kompilierung, sowie die G-Maschine und die STG-Maschine, vor.

Kapitel 3 befaßt sich mit dem von uns implementierten STG-Interpreter, mit dem wir analysieren können, wie ein funktionales Programm durch die STG-Maschine ausgewertet wird. Wir verwenden diesen STG-Interpreter in Kapitel 5 zum Vergleich der Auswertung eines funktionalen Programms durch die STG-Maschine und die G-Maschine. Ein Interpreter für die G-Maschine wurde in [PJL91] vorgestellt.

In **Kapitel 4** optimieren wir die STG-Maschine durch die Einführung von expliziten Registermanipulationsbefehlen. Die STG-Maschine führte bisher alle Berechnungen unter Verwendung von Stacks aus. Dabei wurde außer Acht gelassen, daß heutige Hardwarearchitekturen über viele Register verfügen. Wenn wir die STG-Maschine stärker an der Hardware orientieren und anstelle von Stacks Register verwenden, können wir viele Operationen effizienter ausführen.

Dazu erweitern wir die STG-Sprache um explizite Registermanipulationsbefehle. Zur automatischen Erzeugung des STG-Codes mit expliziten Registermanipulationsbefehlen implementieren wir eine Filter-Funktion, die ein STG-Programm (ohne Registermanipulationsbefehle) in ein STG-Programm mit Registermanipulationsbefehlen übersetzt.

Dieses STG-Programm mit Registermanipulationsbefehlen nehmen wir als Eingabe für den STG-Kompiler von Frank Kiesel, den wir so erweitert haben, daß er die Registermanipulationsbefehle korrekt übersetzt. Der STG-Kompiler erzeugt ein C-Programm mit Registermanipulationsbefehlen, welches wir in Maschinensprache übersetzen können.

Führen wir ein Maschinensprache-Programm aus, das aus STG-Code mit Registermanipulationsbefehlen erzeugt wurde, so kann dieses in Zeit- und Speicherbedarf effizienter sein, als das Maschinensprache-Programm, das aus dem entsprechenden STG-Code ohne Registermanipulationsbefehle erzeugt wurde.

In **Kapitel 5** vergleichen wir die STG-Maschine und die G-Maschine:

- Die STG-Maschine schreibt flache Applikationen vor (Funktionsanwendung nur auf Variablen oder Konstanten). In der G-Maschine ist die Verwendung von flachen Applikationen freigestellt. Wenn wir von der G-Maschine ein Programm mit flachen Applikationen ausführen lassen, so wird dieses nicht effizienter ausgeführt, als das gleiche Programm ohne flache Applikationen.

  Dies liegt daran, daß die G-Maschine intern mit flachen Applikationen arbeitet. Die Maschinen sind in diesem Punkt, obwohl sie verschieden erscheinen, ähnlich.

- Die STG-Maschine verwendet *Updates in Place*, die G-Maschine *Updates durch Indirektionsknoten*. Verändern wir die G-Maschine so, daß sie anstatt *Updates durch Indirektionsknoten Updates in Place* benutzt, so läuft die G-Maschine in Bezug auf Zeit und Platz effizienter.

  Wir können die STG-Maschine in diesem Zusammenhang als optimierte G-Maschine betrachten.

- Um die Auswertung des gleichen (STG-)Programms auf der STG- und der G-Maschine vergleichen zu können, erstellen wir einen STG-To-G-Übersetzer.

  Um das von STG-Code nach G-Code übersetzte Programm ausführen zu können, müssen wir die G-Maschine leicht modifizieren, da diese (im Gegensatz zur STG-Maschine) keine Default-Alternativen von *case*-Anweisungen kennt.

- Die STG-Maschine benutzt, im Gegensatz zur G-Maschine, Updateinformation. Wir modifizieren die G-Maschine so, daß sie die vom STG-To-G-Übersetzer gelieferte Updateinformation auch verwendet. Damit können wir die Effzienz der G-Maschine nochmals verbessern.

  Auch hier stellt sich die STG-Maschine als optimierte G-Maschine dar.

- Ein Vergleich der Auswertung des gleichen STG-Programms von der STG-Maschine und der G-Maschine, sowie die Gegenüberstellung der operationalen Semantiken beider Maschinen, führt uns zu dem Ergebnis, daß die STG-Maschine die gleichen Konzepte verwendet wie die G-Maschine, um ein Programm auszuwerten.

- Ein Vergleich der Implementierungsmöglichkeiten beider Maschinen zeigt, daß wir die G-Maschine mehr mit der Code-Komponente arbeiten lassen müssen, wenn wir diese auch mit Registermanipulationsbefehlen optimieren wollen.

Wir können in diesem Zusammenhang die STG-Maschine wieder als optimierte G-Maschine sehen, da die STG-Maschine hauptsächlich mit der Code-Komponente arbeitet.

In **Kapitel 6** stellen wir weitere Maschinen vor:

Die ABC-Maschine, die der STG-Maschine ähnlich ist, da sie z.B. zwei Stacks verwendet.

Die *Three-Instruction-Machine*, die auf der G-Maschine aufbaut, aber keinen *Spine* hat und flache Applikationen vorschreibt.

Anhang A

Der Programmcode

A.1 Der STG-Maschinen Interpreter

A.1.1 Datentypen zur Darstellung der STG-Programme

```
> data StgExpr
>  = StgLet Binds StgExpr -- Let expressions
>  | StgLetrec Binds StgExpr -- Letrec expressions
>  | StgCase StgExpr Alts -- Case expressions
>  | StgApp Var Atoms -- Application
>  | StgCons Constr Atoms -- saturated Constructor
>  | StgBiOp Prim Atoms -- saturated built-in op
>  | StgLiteral Int     -- primitive Integer

> type Var = String
> type Vars = [Var]

> type Arg = (Stack,Var)
> type Args = [Arg]

> type Constr = Int
> data Prim = Plus | Minus | Mul | Div | Mod | Eq
>             | Lt | Le | Gt | Ge | Ne | Sqrt
> type Literal = Int
> data Update = Update | NoUpdate

> type Bind = (Var,LambdaForm)
> type Binds = [Bind]
```

```
> type LambdaForm = (Vars,Update,Args,StgExpr)
>
> data Alts = Algebraic [Aalt]
>            |Primitive [Palt]

> data Aalt = AlgebraicAlt (Constr,Vars,StgExpr)
>             | AlgebraicDef Default

> data Palt = PrimitiveAlt (Literal,StgExpr)
>             | PrimitiveDef Default

> data Default = DefaultVar Var StgExpr
>               |DefaultNoVar StgExpr

> data Atom = Literal Int | Variable Var
> type Atoms = [Atom]

> type StgProgram = Binds
```

A.1.2 Datentypen für die STG-Maschine

```
> type StgState = (StgOutput,
>                  StgEvalStack,
>                  StgCode,
>                  StgAs,
>                  StgBs,
>                  StgRegs,
>                  StgRs,
>                  StgUs,
>                  StgHeap,
>                  GlobalEnvm)

> getOutput :: StgState -> StgOutput
> getOutput (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = out

> putOutput :: StgOutput -> StgState -> StgState
> putOutput out' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out',es,code,as,bs,regs,rs,us,heap,globals)
```

```
> getEvalStack :: StgState -> StgEvalStack
> getEvalStack (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = es

> putEvalStack :: StgEvalStack -> StgState -> StgState
> putEvalStack es' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es',code,as,bs,regs,rs,us,heap,globals)

> getCode :: StgState -> StgCode
> getCode (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = code

> putCode :: StgCode -> StgState -> StgState
> putCode code' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code',as,bs,regs,rs,us,heap,globals)

> getAs :: StgState -> StgAs
> getAs (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = as

> putAs :: StgAs -> StgState -> StgState
> putAs as' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code,as',bs,regs,rs,us,heap,globals)

> getBs :: StgState -> StgBs
> getBs (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = bs

> putBs :: StgBs -> StgState -> StgState
> putBs bs' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code,as,bs',regs,rs,us,heap,globals)

> getRegs :: StgState -> StgRegs
> getRegs (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = regs

> putRegs :: StgRegs -> StgState -> StgState
> putRegs regs' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code,as,bs,regs',rs,us,heap,globals)

> getRs :: StgState -> StgRs
> getRs (out,es,code,as,bs,regs,rs,us,heap,globals)
```

```
>  = rs

> putRs :: StgRs -> StgState -> StgState
> putRs rs' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code,as,bs,regs,rs',us,heap,globals)

> getUs :: StgState -> StgUs
> getUs (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = us

> putUs :: StgUs -> StgState -> StgState
> putUs us' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code,as,bs,regs,rs,us',heap,globals)

> getHeap :: StgState -> StgHeap
> getHeap (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = heap

> putHeap :: StgHeap -> StgState -> StgState
> putHeap heap' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code,as,bs,regs,rs,us,heap',globals)

> getEnvm :: StgState -> GlobalEnvm
> getEnvm (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = globals

> putEnvm :: GlobalEnvm -> StgState -> StgState
> putEnvm globals' (out,es,code,as,bs,regs,rs,us,heap,globals)
>  = (out,es,code,as,bs,regs,rs,us,heap,globals')

> data Value = Adressvalue Adr | Integervalue Int

> data ES = Entry Value | RP
> type StgEvalStack = [ES]

> type StgAs = [Adr]

> type StgBs = [Int]

> type StgRegs = Atoms

> type StgHeap = Heap Closure
```

```
> data Closure = HeapItem LambdaForm [Value]

> data Stack = A | B
> type Stacks = [Stack]

> type GlobalEnvm = Assoc Var (Adr,Stacks)
> type LocalEnvm1 = Assoc Var Value
> type LocalEnvm2 = Assoc Var (Adr,Stacks)

> type StgCode = [Instruction]

> type StgOutput = Iseq

> data Instruction = Eval StgExpr LocalEnvm1 LocalEnvm2
>                   |SEval LocalEnvm1 LocalEnvm2
>                   |Enter Adr LocalEnvm1 LocalEnvm2
>                   |ReturnCon LocalEnvm1 LocalEnvm2
>                   |ReturnInt LocalEnvm1 LocalEnvm2

> type StgRs = [Continuation]

> type Continuation = (Alts,LocalEnvm1,LocalEnvm2)

> type StgUs = [Updateframe]

> type Updateframe = (StgAs,StgBs,StgRs,Adr)
```

A.1.3 Hilfsfunktionen

```
> val' :: LocalEnvm1 -> LocalEnvm2 -> GlobalEnvm -> Atom -> Value
> val' p1 p2 s (Literal k) = (Integervalue k)
> val' p1 p2 s (Variable v)
>  | elem v (aDomain p1)
>     = (aLookup p1 v (error" "))
>  | elem v (aDomain p2)
>     = (Adressvalue (fst (aLookup p2 v (error""))))
>  | otherwise
>     = (Adressvalue(fst(aLookup s v
>        (error"Var in no Environment"))))
```

```
> val :: LocalEnvm1 -> LocalEnvm2 -> GlobalEnvm
>        -> [Atom] -> [Value]
> val a1 a2 b cs = map (val' a1 a2 b) cs

> isA :: Stack -> Bool
> isA A = True
> isA _ = False

> isB :: Stack -> Bool
> isB B = True
> isB _ = False
```

Die Funktion *split_values* teilt eine Liste von Atomen in zwei Listen, eine für den A und eine für den B-Stack.

```
> split_values :: Var -> Atoms -> LocalEnvm2
>                  -> GlobalEnvm -> (Atoms,Atoms)
> split_values f as p2 g
>  = split_values2 f as sts p2 g
>    where
>    sts | elem f (aDomain p2)
>           = snd (aLookup p2 f (error" "))
>        | elem f (aDomain g )
>           = snd (aLookup g  f (error"f in no Envm"))

> split_values2 :: Var -> Atoms -> Stacks -> LocalEnvm2 ->
>                  GlobalEnvm -> (Atoms,Atoms)
> split_values2 f [] sts p2 g = ([],[])
> split_values2 f (a:as) (s:ss) p2 g
>   | (isA s)
>     = ((a:fst(split_values2 f as ss p2 g)),
>         snd(split_values2 f as ss p2 g))
>   | (isB s)
>     = (fst(split_values2 f as ss p2 g),
>        (a:snd(split_values2 f as ss p2 g)))
```

Die Funktion *split_names* teilt eine Liste von *(Stack, Var)* in zwei Listen vom Typ *Var*, je nach Stack.

```
> split_names :: Args -> (Vars,Vars)
> split_names [] = ([],[])
> split_names ((A,var):names)
>  = (var:fst(split_names(names)),snd(split_names(names)))
> split_names ((B,var):names)
>  = (fst(split_names(names)),var:snd(split_names(names)))
```

Die Funktion *mix* mischt zwei Listen zu einer nach dem Muster von *Args*.

```
> mix :: [Int] -> [Int] -> Args -> [Int]
> mix as bs [] = []
> mix (a:as) (b:bs) ((A,var):args)
>  = a:(mix as (b:bs) args)
> mix (a:as) (b:bs) ((B,var):args)
>  = b:(mix (a:as) bs args)
```

Die Funktion *stacks* erzeugt eine Liste, in der steht, auf welchem Stack welcher Wert steht.

```
> stacks :: Args -> Stacks
> stacks [] = []
> stacks ((s,v):args) = s:stacks(args)

> vartoatom :: Var -> Atom
> vartoatom v = Variable v

> inttoatom :: Int -> Atom
> inttoatom k = Literal k

> atomtoint :: Atom -> Int
> atomtoint (Literal k) = k
> atomtoint _ = error "trying to convert a non-Literal to an Int."

> atomtovalue :: Atom -> Value
> atomtovalue (Literal k) = (Integervalue k)
> atomtovalue _ = error "trying to convert a var. to a value"

> isAdressvalue :: Value -> Bool
> isAdressvalue (Adressvalue _) = True
```

```
> isAdressvalue _ = False

> isIntegervalue :: Value -> Bool
> isIntegervalue (Integervalue _ ) = True
> isIntegervalue _ = False

> valuetoAdr :: Value -> Adr
> valuetoAdr (Adressvalue a) = a
> valuetoAdr _
>  = error "trying to convert a non-Adress value to an Adress."

> valuetoprimitive :: Value -> Int
> valuetoprimitive (Integervalue k) = k
> valuetoprimitive _
>  = error "trying to convert a non-primitive to a primitive."

> valuetoint:: Value -> StgState -> Int
> valuetoint (Integervalue i) state = i
> valuetoint (Adressvalue a) state = k
>               where
>               (HeapItem lambdaform wsf)
>                = hLookup (getHeap state) a
>               (vs,update,xs,e) = lambdaform
>               k=getintfromexpr e

> inttovalue :: Int -> Value
> inttovalue k = (Integervalue k)

> getintfromexpr :: StgExpr -> Int
> getintfromexpr (StgLiteral k) = k
> getintfromexpr _
>  = error"let or letrec expression must be evaluated first."

> isupdate :: Update -> Bool
> isupdate Update = True
> isupdate NoUpdate = False
```

Die Funktion *isdefaultnovar* gibt den Wert *True* zurück, wenn der Default-Fall keine Variable bindet (sondern das Schlüsselwort *default* auftritt).

```
> isdefaultnovar :: Default -> Bool
```

```
> isdefaultnovar (DefaultNoVar expr) = True
> isdefaultnovar _ = False

> isdefaultvar :: Default -> Bool
> isdefaultvar (DefaultVar var expr) = True
> isdefaultvar _ = False
```

A.1.4 Der Startzustand für die STG-Maschine

```
> compile :: StgProgram -> StgState
> compile program = (iStr "",[],initialCode,[],[],[],[],[],
>                    initialHeap,initialGlobals)
>   where
>   initialCode = [Eval (StgApp "main" []) [] [] ]
>   (initialHeap,initialGlobals) = buildinitialHeap program
```

Die Funktion *buildinitialHeap* erzeugt den Anfangs-Heap und die anfängliche *GlobalEnvm* für das STG-Programm.

```
> buildinitialHeap :: StgProgram -> (StgHeap,GlobalEnvm)
> buildinitialHeap program
>  =mapAccuml allocateSc hInitial program
```

Die Funktion *allocateSc* wird für jeden *Bind* aufgerufen und erstellt für jeden *Bind* einen Heap-Knoten.

```
> allocateSc :: StgHeap -> Bind -> (StgHeap,(Var,(Adr,Stacks)))
> allocateSc heap (name,lambdaform)
>  = (heap',globals')
>    where
>    (heap',adr) = hAlloc heap (HeapItem lambdaform [])
>    (vars,update,args,expr)=lambdaform
>    sts = stacks args
>    globals' = (name,(adr,sts))
```

A.1.5 Die Umsetzung der operationalen Semantik

Die Rahmenfunktion: eval

Die *Eval*-Funktion bringt die STG-Maschine zum Laufen.

```
> eval :: StgState -> [StgState]
> eval state = state : restStates
>              where
>              restStates | stgFinal state = []
>                         | otherwise = eval nextState
>              nextState = doAdmin(step state)
>
> stgFinal :: StgState -> Bool
> stgFinal s = null (getCode s)

> step :: StgState -> StgState
> step state = dispatch i (putCode is state)
>              where
>              (i:is) = getCode state
>
> doAdmin :: StgState -> StgState
> doAdmin state = state
> -- spaeter koennen hier statistiken gesp. werden.

> dispatch :: Instruction -> StgState -> StgState
> dispatch (Eval stgExpr localEnvm1 localEnvm2)
>  = stgeval stgExpr localEnvm1 localEnvm2
> dispatch (SEval localEnvm1 localEnvm2)
>  = stgseval localEnvm1 localEnvm2
> dispatch (Enter adr localEnvm1 localEnvm2)
>  = stgenter adr localEnvm1 localEnvm2
> dispatch (ReturnCon localEnvm1 localEnvm2)
>  = stgreturncon localEnvm1 localEnvm2
> dispatch (ReturnInt localEnvm1 localEnvm2)
>  = stgreturnint localEnvm1 localEnvm2
> dispatch _ = error"unknown dispatch-Argument."
```

Die Zustandsübergänge für den stgeval-Befehl

```
> stgeval :: StgExpr -> LocalEnvm1 -> LocalEnvm2
>             -> StgState -> StgState

> stgeval (StgLiteral k) p1 p2 state
>  = putCode ((ReturnInt p1 p2):code) (putRegs [Literal k] state)
>    where
>    code=getCode state

> stgeval (StgApp f xs) p1 p2 state
>  | (isIntegervalue x)    =  rule15 (StgApp f xs) p1 p2 state
>  | (elem f (aDomain p1)) =  rule3 (StgApp f xs) p1 p2 state
>  | otherwise             =  rule2 (StgApp f xs) p1 p2 state
>    where
>    x = val' p1 p2 (getEnvm state) (Variable f)

> stgeval(StgLet binds e) p1 p2 state
>  =  rule7a (StgLet binds e) p1 p2 state

> stgeval(StgLetrec binds e) p1 p2 state
>  =  rule7b (StgLetrec binds e) p1 p2 state

> stgeval(StgCase e alts) p1 p2 state
>  = putCode ((Eval e p1 p2):code) (putRs((alts,p1,p2):rs) state)
>    where
>    code = getCode state
>    rs = getRs state

> stgeval(StgCons c xs) p1 p2 state
>  = putCode ((ReturnCon p1 p2):code)
>             (putRegs ((Literal c):xs) state)
>    where
>    code = getCode state

> stgeval(StgBiOp op [(Variable x1),(Variable x2)]) p1 p2 state
>  = stgeval(StgBiOp op [(Literal i1),(Literal i2)]) p1 p2 state
>    where
>    i1 = valuetoint value1 state
>    i2 = valuetoint value2 state
>    value1 = (aLookup p1 x1 (error"fehler"))
>    value2 = (aLookup p1 x2 (error"fehler"))

> stgeval(StgBiOp op [(Literal i1),(Variable x2)]) p1 p2 state
>  = stgeval(StgBiOp op [(Literal i1),(Literal i2)]) p1 p2 state
```

```
>    where
>    value2 = (aLookup p1 x2 (error"fehler"))
>    i2 = valuetoint value2 state
>    (Adressvalue a) = value2

> stgeval(StgBiOp op [(Variable x1),(Literal i2)]) p1 p2 state
>  = stgeval(StgBiOp op [(Literal i1),(Literal i2)]) p1 p2 state
>    where
>    value1 = (aLookup p1 x1 (error"fehler"))
>    i1 = valuetoint value1 state
>    (Adressvalue a) = value1

> stgeval(StgBiOp Plus [(Literal i1),(Literal i2)]) p1 p2 state
>  = putCode((ReturnInt p1 p2):getCode(state))
>           (putRegs [Literal (i1+i2)] state)
> stgeval(StgBiOp Minus [(Literal i1),(Literal i2)]) p1 p2 state
>  = putCode((ReturnInt p1 p2):getCode(state))
>           (putRegs [Literal (i1-i2)] state)
> stgeval(StgBiOp Mul [(Literal i1),(Literal i2)]) p1 p2 state
>  = putCode((ReturnInt p1 p2):getCode(state))
>           (putRegs [Literal (i1*i2)] state)
> stgeval(StgBiOp Div [(Literal i1),(Literal i2)]) p1 p2 state
>  = putCode((ReturnInt p1 p2):getCode(state))
>           (putRegs [Literal (i1/i2)] state)
> stgeval(StgBiOp Mod [(Literal i1),(Literal i2)]) p1 p2 state
>  =  putCode((ReturnInt p1 p2):getCode(state))
>            (putRegs [Literal (mod i1 i2)] state)
> stgeval(StgBiOp Eq [(Literal i1),(Literal i2)]) p1 p2 state
>  | (i1 == i2) = putCode((ReturnInt p1 p2):getCode(state))
>                         (putRegs[Literal 1] state)
> -- 1 = gleich
>  | otherwise  = putCode((ReturnInt p1 p2):getCode(state))
>                         (putRegs [Literal 0] state)
> -- 0 = ungleich
> stgeval(StgBiOp Le [(Literal i1),(Literal i2)]) p1 p2 state
>  | (i1 <= i2) = putCode((ReturnInt p1 p2):getCode(state))
>                         (putRegs [Literal 1] state)
> -- 1 = '<='
>  | otherwise  = putCode((ReturnInt p1 p2):getCode(state))
>                         (putRegs [Literal 0] state)
> -- 0 = sonst
> stgeval(StgBiOp Lt [(Literal i1),(Literal i2)]) p1 p2 state
>  | (i1 <  i2) = putCode((ReturnInt p1 p2):getCode(state))
```

```
>                           (putRegs [Literal 1] state)
> -- 1 = '<'
>  | otherwise  = putCode((ReturnInt p1 p2):getCode(state))
>                           (putRegs [Literal 0] state)
> -- 0 = sonst
> stgeval(StgBiOp Ge [(Literal i1),(Literal i2)]) p1 p2 state
>  | (i1 >= i2) = putCode((ReturnInt p1 p2):getCode(state))
>                           (putRegs [Literal 1] state)
> -- 1 = '>='
>  | otherwise  = putCode((ReturnInt p1 p2):getCode(state))
>                           (putRegs [Literal 0] state)
> -- 0 = sonst
> stgeval(StgBiOp Gt [(Literal i1),(Literal i2)]) p1 p2 state
>  | (i1 >  i2) = putCode((ReturnInt p1 p2):getCode(state))
>                           (putRegs [Literal 1] state)
> -- 1 = '>'
>  | otherwise  = putCode((ReturnInt p1 p2):getCode(state))
>                           (putRegs [Literal 0] state)
> -- 0 = sonst
> stgeval(StgBiOp Ne [(Literal i1),(Literal i2)]) p1 p2 state
>  | (i1 /= i2) = putCode((ReturnInt p1 p2):getCode(state))
>                           (putRegs [Literal 1] state)
> -- 1 = ungleich
>  | otherwise  = putCode((ReturnInt p1 p2):getCode(state))
>                           (putRegs [Literal 0] state)
> -- 0 = gleich
> stgeval(StgBiOp Sqrt[(Literal i1),(Literal i2)]) p1 p2 state
>  = putCode((ReturnInt p1 p2):getCode state)
>              (putRegs [Literal (truncate (sqrt
>                         (primIntToFloat i1)))] state)

> stgeval expr p1 p2 state = error "stgeval: unknown expression."
```

Die Funktion *rule3* ist ein Spezialfall von *stgeval (StgApp f xs)*. *f* ist dann Element von ρ_1.

```
>  rule3 :: StgExpr -> LocalEnvm1 -> LocalEnvm2
>              -> StgState -> StgState
>  rule3 (StgApp f xs) p1 p2 state
>    = putCode ((Enter a p1 p2):code) (putAs (as'++as) state)
>    where
```

```
>    code=getCode state
>    as = getAs state
>    as1 = val p1 p2 (getEnvm state) xs
>    as' | (null as1) = []
>        | otherwise  = map valuetoAdr as1
>    a1 = val' p1 p2 (getEnvm state) (vartoatom f)
>    a = valuetoAdr a1
```

Die Funktion *rule2* ist ein Spezialfall von *stgeval (StgApp f xs)*. *f* ist dann Element von ρ_2 oder σ. Außerdem zeigt *f* auf eine Adresse.

```
>  rule2 :: StgExpr -> LocalEnvm1 -> LocalEnvm2
>            -> StgState -> StgState
>  rule2 (StgApp f xs) p1 p2 state
>  = putCode ((Enter a p1 p2):code)
>             (putAs (as''++as) (putBs (bs''++bs) state))
>    where
>    x = val' p1 p2 (getEnvm state) (Variable f)
>    (as',bs')=split_values f xs p2 (getEnvm state)
>    as2=val p1 p2 (getEnvm state) as'
>    as''=map valuetoAdr as2
>    bs2=val p1 p2 (getEnvm state) bs'
>    bs''=map valuetoprimitive bs2
>    code = getCode state
>    as = getAs state
>    bs = getBs state
>    (Adressvalue a) = x
```

Die Funktion *rule15* ist ein Spezialfall von *stgeval (StgApp f xs)*. *f* ist dann Element von ρ_2 oder σ. Außerdem zeigt *f* auf einen Integerwert.

```
>  rule15 :: StgExpr -> LocalEnvm1 -> LocalEnvm2
>             -> StgState -> StgState
>  rule15 (StgApp f xs) p1 p2 state
>  = putCode ((ReturnInt p1 p2):code) (putRegs [Literal k] state)
>    where
>    x = val' p1 p2 (getEnvm state) (Variable f)
>    code = getCode state
>    (Integervalue k) = x
```

Die Funktion *rule7a* wandelt das erste *Bind* in einen Heap-Knoten um und ruft sich dann mit den restlichen *Binds* rekursiv auf. (Das ist das zweite Pattern im Programmtext). Sind keine *Binds* mehr vorhanden hört sie auf (erstes Pattern).

```
>  rule7a :: StgExpr -> LocalEnvm1 -> LocalEnvm2
>              -> StgState -> StgState
>  rule7a (StgLet [] e) p1 p2 state
>  = putCode((Eval e p1 p2):(getCode state)) state
>  rule7a (StgLet ((var,lambdaform):binds) e) p1 p2 state
>  =  rule7a (StgLet binds e) p1 p2' (putHeap h' state)
>    where
>    (vs,update,xs,lfe) = lambdaform
>    w = val p1 p2'' (getEnvm state) (map vartoatom vs)
>    (h',a)=hAlloc (getHeap state) (HeapItem lambdaform w)
>    p2' = p2 ++ makeenvm2 [var] [(a,(stacks xs))]
>    p2''=p2

>  rule7b :: StgExpr -> LocalEnvm1 -> LocalEnvm2
>              -> StgState -> StgState
>  rule7b (StgLetrec [] e) p1 p2 state
>  = putCode((Eval e p1 p2):(getCode state)) state
>  rule7b (StgLetrec ((var,lambdaform):binds) e) p1 p2 state
>  =  rule7b (StgLetrec binds e) p1 p2' (putHeap h' state)
>    where
>    (vs,update,xs,lfe) = lambdaform
>    w = val p1 p2'' (getEnvm state) (map vartoatom vs)
>    (h',a)=hAlloc (getHeap state) (HeapItem lambdaform w)
>    p2' = p2 ++ makeenvm2 [var] [(a,(stacks xs))]
>    p2''=p2'
```

Die Zustandsübergänge für den stgseval-Befehl

```
> stgseval :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> stgseval e1 e2 state
>  | (null (getEvalStack state))
>     = state -- wir sind fertig.
>  | (isIntegerEntry top)
>     = stgseval_case1 e1 e2 state -- top schon evaluiert
>  | (isAdressEntry top)
```

```
>     = stgseval_case2 e1 e2 state -- top noch nicht evaluiert
>   | (isRP top)
>     = stgseval_case3 e1 e2 state -- Klammer zu
>   | otherwise = error"undefined constellation in stgseval!"
>    where
>    top = head(getEvalStack state)

> isIntegerEntry :: ES -> Bool
> isIntegerEntry (Entry (Integervalue i)) = True
> isIntegerEntry _ = False

> isAdressEntry :: ES -> Bool
> isAdressEntry (Entry (Adressvalue a)) = True
> isAdressEntry _ = False

> isRP :: ES -> Bool
> isRP RP = True
> isRP _ = False

> stgseval_case1 :: LocalEnvm1 -> LocalEnvm2
>                    -> StgState -> StgState
> -- top schon evaluiert (Integervalue).
> stgseval_case1 e1 e2 state
>   | (null rest) = putOutput(out_new) (putEvalStack [] state)
>   | otherwise = putOutput(out_new) (putEvalStack rest
>                (putCode ((SEval e1 e2):(getCode state)) state))
>    where
>    top = head(getEvalStack state)
>    (Entry (Integervalue i)) = top
>    rest = tail(getEvalStack state)
>    out_old = getOutput state
>    out_new = iConcat[out_old,iStr(show i)]

> stgseval_case2 :: LocalEnvm1 -> LocalEnvm2
>                    -> StgState -> StgState
> -- top noch nicht evaluiert (Adressvalue)
> stgseval_case2 e1 e2 state
>   = putEvalStack rest (putCode ((Eval e e1 e2):code) state)
>    where
>    code=getCode state
>    top=head(getEvalStack state)
>    rest=tail(getEvalStack state)
>    (Entry(Adressvalue a))=top
```

```
>    (HeapItem lambdaform values) = hLookup (getHeap state) a
>    (fvars,update,avars,e) = lambdaform

> stgseval_case3 :: LocalEnvm1 -> LocalEnvm2
>                    -> StgState -> StgState
> -- Klammer zu
> stgseval_case3 e1 e2 state
>  = putOutput(out_new) (putEvalStack rest
>     (putCode ((SEval e1 e2):(getCode state)) state))
>    where
>    rest = tail (getEvalStack state)
>    out_old = getOutput state
>    out_new = iConcat[out_old,iStr"}"]
```

Die Zustandsübergänge für den stgenter-Befehl

```
> stgenter :: Adr -> LocalEnvm1 -> LocalEnvm2
>              -> StgState -> StgState
> stgenter a e1 e2 state
>  | ((not(isupdate(update)))&&((length(as)>=length(as'))
>      &&(length(bs)>=length(bs'))))
>    =  rule5 a e1 e2 state
>  | ((not(isupdate(update)))&&((length(as)<length(as'))
>      ||(length(bs)<length(bs'))))
>    =  rule6 a e1 e2 state
>  | ((isupdate(update))&&(null(xs)))
>    =  rule4 a e1 e2 state
>  | otherwise =
>    error "closures with non-empty arg.list are never updatable"
>   where
>   as = getAs state
>   bs = getBs state
>   heap = getHeap state
>   (HeapItem lambdaform wsf) = hLookup heap a
>   (vs,update,xs,e) = lambdaform
>   (as',bs') = split_names xs
```

Die Funktion *rule4* ist eine Spezialfall von *stgenter*. Die *Closure* ist dann *updatable* und es gibt keine Argumente.

```
> rule4 :: Adr -> LocalEnvm1 -> LocalEnvm2
>           -> StgState -> StgState
> rule4 a e1 e2 state
> = putCode ((Eval e (doppelteraus (p1++e1)) e2):code)
>            (putAs [] (putBs [] (putRs [] (putUs us' state))))
> where
> code = getCode state
> heap = getHeap state
> (HeapItem lambdaform wsf) = hLookup heap a
> (vs,update,xs,e) = lambdaform
> p1 = makeenvm1 vs wsf
> us = getUs state
> rs = getRs state
> as = getAs state
> bs = getBs state
> us' = ((as,bs,rs,a):us)
```

Die Funktion *rule5* ist eine Spezialfall von *stgenter*. Die *Closure* ist dann nicht-*updatable* und es sind genug Argumente auf den Stacks.

```
> rule5 :: Adr -> LocalEnvm1 -> LocalEnvm2
>           -> StgState -> StgState
> rule5 a e1 e2 state
> = putCode ((Eval e (doppelteraus (p1++e1)) e2):code)
>            (putAs as' (putBs bs' state))
>   where
>   code = getCode state
>   as = getAs state
>   bs = getBs state
>   heap = getHeap state
>   (HeapItem lambdaform wsf) = hLookup heap a
>   (vs,update,xs,e) = lambdaform
>   (xsa,xsb)=split_names xs
>   numberofargsfora = length(xsa)
>   numberofargsforb = length(xsb)
>   wsa = take numberofargsfora as
>   as' = drop numberofargsfora as
>   wsb = take numberofargsforb bs
>   bs' = drop numberofargsforb bs
>   p1 = makeenvm1 vs wsf ++ makeenvm1 xsa (map inttovalue wsa)
```

```
>         ++ makeenvm1 xsb (map inttovalue wsb)

> doppelteraus :: LocalEnvm1 -> LocalEnvm1
> doppelteraus [] = []
> doppelteraus ((x,v):rest)
>  = ((x,v):doppelteraus(alleraus x rest))

> alleraus x [] = []
> alleraus x ((y,s):rest) | (x==y) = alleraus x rest
>                         | otherwise = ((y,s):alleraus x rest)
```

Die Funktion *rule6* ist eine Spezialfall von *stgenter*. Die *Closure* ist nicht-*updatable* und es sind nicht genug Argumente auf den Stacks.

```
>  rule6 :: Adr -> LocalEnvm1 -> LocalEnvm2
>           -> StgState -> StgState
>  rule6 a e1 e2 state
>  = putCode ((Enter a e1 e2):code) (putAs as'
>             (putBs bs'(putRs rsu(putUs us(putHeap hu state)))))
>    where
>    code = getCode state
>    ((asu,bsu,rsu,au):us) = getUs state
>    as = getAs state
>    as' = as++asu
>    bs = getBs state
>    bs' = bs ++ bsu
>    numberofvars = length(as) + length(bs)
>    ys = makevars numberofvars
>    (HeapItem lambdaform wsf) = hLookup (getHeap state) a
>    (vs,update,xs,e) = lambdaform
>    e' = (StgApp f (map vartoatom ys))
>    f = "was"
>    lambdaform' = ((f:ys),NoUpdate,[],e')
>    wf = map inttovalue (a:(mix as bs xs))
>    hu = hUpdate (getHeap state) au (HeapItem lambdaform' wf)

> makeenvm1 :: [Var] -> [Value] -> LocalEnvm1
> makeenvm1 [] [] = []
> makeenvm1 [] bs = [] -- Variable list empty
> makeenvm1 as [] = [] -- Value list emtpy
> makeenvm1 (a:as) (b:bs) = [(a,b)]++makeenvm1 as bs
```

```
> makeenvm1 _ _ = error "makeenvm: cannot create environment"

> makeenvm2 :: [Var] -> [(Adr,Stacks)] -> LocalEnvm2
> makeenvm2 [] [] = []
> makeenvm2 [] bs = [] -- Variable list is empty
> makeenvm2 as [] = [] -- Value list is emtpy
> makeenvm2 (a:as) (b:bs) = [(a,b)]++makeenvm2 as bs
> makeenvm2 _ _
>  = error "makeenvm: cannot create environment"
```

Die Zustandsübergänge für den ReturnCon-Befehl

```
> stgreturncon :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> stgreturncon e1 e2 state
>     | ((null rs) && (null us)) = constructor_out e1 e2 state
>     -- empty RS and empty US: output.
>     | (null rs) =  rule13 e1 e2 state
>     -- Return-Stack is empty.
>     | (isamatch c aalts) =  rule10 e1 e2 state
>     -- constr. matches an alt.
>     | (isdefaultnovar defaultalt) =  rule11 e1 e2 state
>     -- defaultalt & no var.
>     | (isdefaultvar defaultalt)  =  rule12 e1 e2 state
>     -- defaultalt & variable.
>     | otherwise =
>       error "some strange constellation occured in stgreturncon."
>       where
>       rs = getRs state
>       us = getUs state
>       ((alts,p1,p2):rs2) = rs
>       (Algebraic aalts) = alts
>       defaultalt = getadefaultalt aalts
>       regs = getRegs state
>       c = atomtoint(head regs)
>       ws = tail regs

> getadefaultalt :: [Aalt] -> Default
> getadefaultalt ((AlgebraicDef defaultalt):ss) = defaultalt
> getadefaultalt (s:ss) = getadefaultalt ss
> getadefaultalt _
```

```
>  = error "trying to get a defaultalt but there is no"
```

Die Funktion *constructor_out* gibt den Konstruktor aus und erzwingt die Auswertung evtl. vorhandener Konstruktor-Argumente.

```
> constructor_out :: LocalEnvm1 -> LocalEnvm2
>                    -> StgState -> StgState
> constructor_out e1 e2 state
>  = putOutput (constructor) (putEvalStack
>     ((atomstoentry e1 e2 state ws)
>      ++ [RP] ++ oldEvalStack)
>      (putCode ((SEval e1 e2):(getCode state)) state))
>    where
>    oldEvalStack=getEvalStack state
>    out_old = getOutput state
>    constructor=iConcat[out_old,iStr"{<",iStr(show c),iStr"> "]
>    regs=getRegs state
>    c = atomtoint (head regs)
>    ws = tail regs

> atomstoentry :: LocalEnvm1 -> LocalEnvm2
>                 -> StgState -> [Atom] -> [ES]
> atomstoentry e1 e2 state [] = []
> atomstoentry e1 e2 state ((Literal i):as) =
>  ((Entry (Integervalue i)):(atomstoentry e1 e2 state as))
> atomstoentry e1 e2 state ((Variable v):as) =
>  ((Entry (val' e1 e2 globals (Variable v)):
>   (atomstoentry e1 e2 state as)))
>  where
>  globals=getEnvm state
```

Die Funktion *isamatch* liefert den Wert *True*, wenn der Konstruktor eine der Alternativen auswählt.

```
> isamatch :: Constr -> [Aalt] -> Bool
> isamatch c ((AlgebraicAlt (c1,vars,stgexpr)):ss)
>  | (c==c1) = True
>  | otherwise = isamatch c ss
> isamatch c _ = False
```

Die Funktion *rule10* ist ein Spezialfall von *ReturnCon*. Der Konstruktor wählt eine der Alternativen aus.

```
> rule10 :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> rule10 e1 e2 state
>        = putCode((Eval e p1' p2):code) (putRs rs state)
>       where
>       regs = getRegs state
>       c=atomtoint(head regs)
>       ws = tail regs
>       code = getCode state
>       ((alts,p1,p2):rs) = getRs state
>       (Algebraic aalts) = alts
>       (vs,e) = aaltslookup aalts c
>       ws2 = val e1 e2 (getEnvm state) ws
>       p1' = p1++makeenvm1 vs ws2
```

Die Funktion *rule11* ist ein Spezialfall von *ReturnCon*. Die Default-Alternative muß verwendet werden, da der Konstruktor keine Alternative auswählt. Außerdem hat die Default-Alternative keine Variable, die gebunden wird.

```
> rule11 :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> rule11 e1 e2 state
> = putCode((Eval ed p1 p2):code) (putRs rs state)
>       where
>       regs = getRegs state
>       c=head regs
>       ws = tail regs
>       code = getCode state
>       ((alts,p1,p2):rs) = getRs state
>       (Algebraic aalts) = alts
>       defaultalt = getadefaultalt aalts
>       (DefaultNoVar ed) = defaultalt
```

Die Funktion *rule12* ist eine Spezialfall von ReturnCon. Die Default-Alternative muß verwendet werden, da der Konstruktor keine Alternative auswählt. Die Default-Alternative hat eine Variable, die gebunden werden soll.

```
> rule12 :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> rule12 e1 e2 state
> = putCode((Eval ed p1' p2):code) (putRs rs (putHeap h' state))
>    where
>    regs=getRegs state
>    c=atomtoint(head regs)
>    ws=tail regs
>    code = getCode state
>    ((alts,p1,p2):rs) = getRs state
>    (Algebraic aalts) = alts
>    defaultalt = getadefaultalt aalts
>    (DefaultVar v ed) = defaultalt
>    heap = getHeap state
>    numberofvars = length(ws)
>    vs = makevars numberofvars
>    lambdaform =
>     (vs,NoUpdate,[],(StgCons c (map vartoatom vs)))
>    (h',a) =
>     hAlloc heap (HeapItem lambdaform (val e1 e2
>                  (getEnvm state) ws))
>    p1' = (makeenvm1 [v] [Adressvalue a]) ++ p1
```

Die Funktion *rule13* ist ein Spezalfall von ReturnCon. Der Return-Stack ist leer.

```
> rule13 :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> rule13 e1 e2 state
> = putCode((ReturnCon e1 e2):code)
>          (putAs asu (putBs bsu(putRs rsu
>          (putUs us (putHeap hu state)))))
>   where
>   regs=getRegs state
>   c=atomtoint(head regs)
>   ws=tail regs
>   ((asu,bsu,rsu,au):us) = getUs state
>   code = getCode state
>   hu = hUpdate (getHeap state) au closure
>   numberofvars = length(ws)
>   vs = makevars numberofvars
>   lf = (vs,NoUpdate,[],(StgCons c (map vartoatom vs)))
>   ws2= val e1 e2 (getEnvm state) ws
>   closure = (HeapItem lf ws2)
```

Die Funktion *makevars n* generiert *n* verschiedene Variablen. Diese Funktion wird von der Funktion *rule13* gebraucht.

```
> makevars :: Int -> Vars
> makevars 0 = []
> makevars n = [copy n 'a'] ++ makevars (n-1)
```

Die Funktion *aaltslookup* wird nur ausgeführt, wenn der Konstruktor eine Alternative auswählt. Die Funktion liefert die Variablen und den STG-Code der ausgewählten Alternative.

```
> aaltslookup :: [Aalt] -> Constr -> ([Var],StgExpr)
> aaltslookup [] c = error "cannot find the matching constructor."
> aaltslookup ((AlgebraicAlt (constr,vs,e)):aalts) c
>              | (constr==c) = (vs,e)
>              | otherwise = aaltslookup aalts c
```

Die Zustandsübergänge für den ReturnInt-Befehl

```
> stgreturnint :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> stgreturnint e1 e2 state
>  | (null(getRs state)) = printint e1 e2 state
>    -- cannot update closure with primitive type.
>  | (ispmatch k palts) =  rule16 state
>    -- the int matches one alternative.
>  | (isdefaultnovar defaultalt) =  rule17 state
>    -- defaultalt with no var.
>  | (isdefaultvar defaultalt) =  rule18 state
>    -- defaultalt with var.
>  | otherwise = error "stgreturnint: no match"
>      where
>      regs = getRegs state
>      k=atomtoint(head regs)
>      code=getCode state
>      ((alts,p1,p2):rs)=getRs state
>      (Primitive palts) = alts
```

```
>      defaultalt = getpdefaultalt palts
```

Die Funktion *ispmatch* liefert True, wenn der Integer-Wert eine Alternative auswählt.

```
> ispmatch :: Int -> [Palt] -> Bool
> ispmatch k ((PrimitiveAlt (k1,e)):ss)
>  | (k==k1) = True
>  | otherwise = ispmatch k ss
> ispmatch k _ = False

> getpdefaultalt ::  [Palt] -> Default
> getpdefaultalt ((PrimitiveDef defaultalt):ss) = defaultalt
> getpdefaultalt (s:ss) = getpdefaultalt ss
> getpdefaultalt _ = error "there is no defaultalt."
```

Die Funktion *printint* druckt einen Integer-Wert aus.

```
> printint :: LocalEnvm1 -> LocalEnvm2 -> StgState -> StgState
> printint e1 e2 state
>  | (null (getEvalStack state))
>    = putOutput (output) state
>  | otherwise
>    = putOutput (output) (putCode ((SEval [] []):
>                         (getCode state)) state)
>    where
>    out1 = getOutput state
>    output = iConcat [out1,iStr(show k)]
>    regs=getRegs state
>    k=atomtoint (head regs)
```

Die Funktion *rule16* wird ausgeführt, wenn der Integer-Wert eine Alternative auswählt.

```
>  rule16 :: StgState -> StgState
>  rule16 state
>  = putCode((Eval e p1 p2):code) (putRs rs state)
```

```
>       where
>       code=getCode state
>       ((alts,p1,p2):rs)=getRs state
>       (Primitive palts) = alts
>       regs = getRegs state
>       k=atomtoint(head regs)
>       e = paltslookup palts k
```

Die Funktion *paltslookup* wird nur ausgeführt, wenn der Integer-Wert eine Alternative auswählt. Die Funktion liefert den STG-Code der ausgewählten Alternative.

```
> paltslookup :: [Palt] -> Int -> StgExpr
> paltslookup [] k = error "no match."
> paltslookup ((PrimitiveAlt (k1,e)):ss) k
>  | (k1==k) = e
>  | otherwise = paltslookup ss k
```

Die Funktion *rule17* wird ausgeführt, wenn der Integer-Wert keine Alternative auswählt, also die Default-Alternative genommen werden muß und die Default-Alternative keine Variable bindet.

```
>  rule17 :: StgState -> StgState
>  rule17 state
>  = putCode((Eval ed p1 p2):code) (putRs rs state)
>       where
>       code=getCode state
>       ((alts,p1,p2):rs)=getRs state
>       (Primitive palts) = alts
>       defaultalt=getpdefaultalt palts
>       (DefaultNoVar ed)=defaultalt
```

Die Funktion *rule18* wird ausgeführt, wenn der Integer-Wert keine Alternative auswählt, also die Default-Alternative genommen werden muß und die Default-Alternative eine Variable bindet.

```
>  rule18 :: StgState -> StgState
```

```
>  rule18 state
>  = putCode((Eval ed p1' p2):code) (putRs rs state)
>      where
>      code=getCode state
>      ((alts,p1,p2):rs)=getRs state
>      (Primitive palts) = alts
>      defaultalt=getpdefaultalt palts
>      (DefaultVar var ed) = defaultalt
>      regs = getRegs state
>      k=atomtoint(head regs)
>      p1' = makeenvm1 [var] [Integervalue k] ++ p1
```

A.1.6 Die Ausgabe-Funktion

```
> data Iseq = INil
>           | IStr String
>           | IAppend Iseq Iseq
>           | IIndent Iseq
>           | INewline

> iNil              = INil
> iAppend seq1 seq2 = IAppend seq1 seq2
> iStr str          = IStr str
> iIndent seq       = IIndent seq
> iNewline          = INewline

> iDisplay :: Iseq -> String
> iDisplay seq = flatten 0 [(seq,0)]

> flatten :: Int
>           -> [(Iseq, Int)]
>           -> String
> flatten col ((INewline, indent) : seqs)
>   = '\n' : (space indent) ++ (flatten indent seqs)
> flatten col ((IIndent seq, indent) : seqs)
>   = flatten col ((seq, col) : seqs)
> flatten col ((IStr s, indent) : seqs)
>   = s ++ flatten (col + length s) seqs
> flatten col ((INil, indent) : seqs) = flatten col seqs
> flatten col ((IAppend seq1 seq2, indent) : seqs)
```

```
>   = flatten col ((seq1, indent) : (seq2, indent) : seqs)
> flatten col [] = ""

> iConcat = foldr iAppend iNil

> iInterleave sep []         = iNil
> iInterleave sep [seq]      = seq
> iInterleave sep (seq:seqs) = seq `iAppend`
>                 (sep `iAppend` iInterleave sep seqs)

> iNum :: Int -> Iseq
> iNum n = iStr (show n)
>
> iFWNum :: Int -> Int -> Iseq
> iFWNum width n
>   = iStr (space (width - length digits) ++ digits)
>     where
>     digits = show n

> iLayn :: [Iseq] -> Iseq
> iLayn seqs = iConcat (map lay_item (zip [1..] seqs))
>              where
>              lay_item (n, seq)
>                = iConcat [ iFWNum 4 n, iStr ") ",
>                   iIndent seq, iNewline ]

> iSpace = iStr " "

> showResults :: [StgState] -> [Char]
> showResults states
>  = iDisplay (iConcat [
>    iStr "Supercombinator definitions",iNewline,
>    iInterleave iNewline (map (showSC s) (getEnvm s)),
>    iNewline, INewline, iStr "State transitions",
>    iNewline,iNewline,
>    iLayn (map showState states),
>    iNewline,iNewline,
>    showStats (last states)])
>    where
>    (s:ss) = states

> showSC :: StgState -> (Var,(Adr,Stacks)) -> Iseq
> showSC s (name,(adr,stacks))
```

```
>  = iConcat [iStr "Code for ",iStr name,INewline,
>    iStr name, iStr " = ",iStr "{",showvars fvars,iStr "} ",
>    showupdate update,iStr " ", iStr " {",
>    showargs avars,iStr "}",
>    iStr " -> ",showInstructions stgexpr,iNewline]
>    where
>    (HeapItem lambdaForm values) = hLookup (getHeap s) adr
>    (fvars,update,avars,stgexpr) = lambdaForm
```

Die Funktion *showvars* gibt eine Liste von Variablen aus.

```
> showvars :: Vars -> Iseq
> showvars [] = iStr ""
> showvars (s:ss)
> | (null ss)
>   = iStr s
> | otherwise
>   = iConcat [iStr s,iStr",",showvars ss]
```

Die Funktion *showargs* gibt eine Liste von Argumenten aus.

```
> showargs :: Args -> Iseq
> showargs [] = iStr ""
> showargs ((stack,var):ss)
>  | ((isA stack)&&(null ss)) = iConcat [iStr "A:",iStr var]
>  | ((isB stack)&&(null ss)) = iConcat [iStr "B:",iStr var]
>  | (isA stack)
>    = iConcat [iStr "A:",iStr var,iStr",",showargs ss]
>  | (isB stack)
>    = iConcat [iStr "B:",iStr var,iStr",",showargs ss]
```

Die Funktion *showupdate* gibt das Update-Flag (von einer Lambdaform) aus.

```
> showupdate :: Update -> Iseq
> showupdate Update = iStr "\\u"
> showupdate NoUpdate = iStr "\\n"
```

Die Funktion *showInstructions* gibt einen STG-Ausdruck aus.

```
> showInstructions :: StgExpr -> Iseq
> showInstructions (StgLet binds stgexpr)
>  = iConcat [iStr "let ",showbinds binds,iStr " in ",
>             iNewline,iStr "  ",showInstructions stgexpr]
> showInstructions (StgLetrec binds stgexpr)
>  = iConcat [iStr "letrec ",showbinds binds,iStr " in ",
>             showInstructions stgexpr]
> showInstructions (StgCase stgexpr alts)
>  = iConcat[iStr "case ",showInstructions stgexpr,
>            iStr " of ",showalts alts]
> showInstructions (StgApp var atoms)
>  = iConcat [iStr var, iStr " ",showatoms atoms]
> showInstructions (StgCons constr atoms)
>  = iConcat[iStr"<",iStr (show constr),
>            iStr"> {",showatoms atoms,iStr"}"]
> showInstructions (StgBiOp prim atoms)
>  = showBiOp prim atoms
> showInstructions (StgLiteral literal)
>  = (iStr (show literal))
> showInstructions _
>  = (iStr "showInstructions: unknown instruction!")
```

Die Funktion *showalts* gibt die Alternativen einer *case*-Anweisung aus.

```
> showalts :: Alts -> Iseq
> showalts (Algebraic aalts)
>  = showaalts aalts
> showalts (Primitive palts)
>  = showpalts palts
```

Die Funktion *showaalts* gibt die Liste der algebraischen Alternativen aus.

```
> showaalts :: [Aalt] -> Iseq
> showaalts [] = iStr ""
> showaalts ((AlgebraicAlt (constr,vars,stgexpr)):ss)
>  = iConcat[iNewline,iStr"    <",iStr(show constr),iStr"> {",
```

```
>             showvars vars,iStr"} ",
>             showInstructions stgexpr,iStr";",showaalts ss]
> showaalts ((AlgebraicDef defaultalt):ss)
>  = showdefaultalt defaultalt
```

Die Funktion *showpalts* gibt die Liste der primitiven Alternativen aus.

```
> showpalts :: [Palt] -> Iseq
> showpalts [] = iStr ""
> showpalts ((PrimitiveAlt (literal,stgexpr)):ss)
>  = iConcat[iNewline,iStr"    ",iStr(show literal),iStr" -> ",
>             showInstructions stgexpr,showpalts ss]
> showpalts ((PrimitiveDef defaultalt):ss)
>  = showdefaultalt defaultalt
```

Die Funktion *showdefaultalt* zeigt die Default-Alternative.

```
> showdefaultalt :: Default -> Iseq
> showdefaultalt (DefaultVar var stgexpr)
>  = iConcat[iNewline,iStr "    ",iStr var,
>             iStr " -> ",showInstructions stgexpr]
> showdefaultalt (DefaultNoVar stgexpr)
>  = iConcat[iNewline,iStr "    ",
>             iStr "default -> ",showInstructions stgexpr]
```

Die Funktion *showbinds* zeigt die *Binds* einer *let* oder *letrec* Anweisung.

```
> showbinds :: Binds -> Iseq
> showbinds [] = iStr ""
> showbinds ((var,lambdaform):ss)
>  = iConcat[iStr var,iStr" = {",showvars fvars,iStr "} ",
>             showupdate update,
>             iStr " ",
>             iStr" {",showargs avars,iStr"} ",iStr " -> ",
>             showInstructions stgexpr]
>             where
>             (fvars,update,avars,stgexpr)=lambdaform
```

Die Funktion *showatoms* zeigt eine Liste von Atomen, z.B. von einer Funktions-Applikation.

```
> showatoms :: Atoms -> Iseq
> showatoms [] = iStr ""
> showatoms (s:ss)
>   | (null ss)
>     = atomtoiStr s
>   | otherwise
>     = iConcat [atomtoiStr s,iStr",",showatoms ss]
```

Die Funktion *atomtoiStr* transformiert ein *Atom* in einen *iStr*, also zum Typen *Iseq*.

```
> atomtoiStr :: Atom -> Iseq
> atomtoiStr (Literal x)
>   = iStr (show x)
> atomtoiStr (Variable x)
>   = iStr x
> atomtoiStr _
>   = error "using a non atom with showatoms"
```

Die Funktion *showBiOp* zeigt eine 'Build-In' Operation, also +,-,*,/ usw.

```
> showBiOp :: Prim -> Atoms -> Iseq
> showBiOp Plus [atom1,atom2]
>   = iConcat[iStr"+ {", atomtoiStr atom1,
>             iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Minus [atom1,atom2]
>   = iConcat[iStr"- {", atomtoiStr atom1,
>             iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Mul [atom1,atom2]
>   = iConcat[iStr"* {", atomtoiStr atom1,
>             iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Div [atom1,atom2]
>   = iConcat[iStr"/ {", atomtoiStr atom1,
>             iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Mod [atom1,atom2]
```

```
>  = iConcat[iStr"% {", atomtoiStr atom1,
>            iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Eq [atom1,atom2]
>  = iConcat[iStr"= {", atomtoiStr atom1,
>            iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Lt [atom1,atom2]
>  = iConcat[iStr"< {", atomtoiStr atom1,
>            iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Le [atom1,atom2]
>  = iConcat[iStr"<= {", atomtoiStr atom1,
>            iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Gt [atom1,atom2]
>  = iConcat[iStr"> {", atomtoiStr atom1,
>            iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Ge [atom1,atom2]
>  = iConcat[iStr">= {", atomtoiStr atom1,
>            iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Ne [atom1,atom2]
>  = iConcat[iStr"!= {", atomtoiStr atom1,
>            iStr",", atomtoiStr atom2, iStr"}"]
> showBiOp Sqrt [atom1,atom2]
>  = iConcat[iStr"sqrt {",atomtoiStr atom1,
>            iStr",",atomtoiStr atom2, iStr"}"]
> showBiOp sonst [atom1,atom2]
>  = iConcat[iStr(show' sonst),
>            iStr" {",atomtoiStr atom1, iStr ",",
>            atomtoiStr atom2,iStr"}"]

> showState :: StgState -> Iseq
> showState s
>  = iConcat [showOutput s,             iNewline,
>             showEvalStack s,          iNewline,
>             showCode1 (getCode s) s , iNewline,
>             showAs s,                 iNewline,
>             showBs s,                 iNewline,
>             showRegs s,               iNewline,
>             showRs s,                 iNewline,
>             showUs s,                 iNewline]

> showOutput :: StgState -> Iseq
> showOutput s
>  = iConcat [iStr "Output:/",getOutput s,iStr "/"]
```

```
> showEvalStack :: StgState -> Iseq
> showEvalStack state
>  | (null es) = iStr "Der Eval-Stack ist leer."
>  | otherwise = iConcat[(iStr "Eval-Stack:"),(iIndent(iInterleave
>               iNewline (map(showEvalStackItem state) es)))]
>   where
>   es = getEvalStack state

> showEvalStackItem :: StgState -> ES -> Iseq
> showEvalStackItem state RP = iStr "}"
> showEvalStackItem state (Entry v) = showStackItem state v

> showCode1 :: StgCode -> StgState -> Iseq
> showCode1 [] s = (iStr "Code: no more Code.")
> showCode1 c s = iConcat [iStr "Code: ",iNewline,showCode c s]

> showCode :: StgCode -> StgState -> Iseq
> showCode((Eval stgExpr localEnvm1 localEnvm2):ss) state
>  = iConcat[iStr "Eval ",showInstructions stgExpr,
>           showlocalEnvm1 localEnvm1 state, showlocalEnvm2
>           localEnvm2 state, showCode ss state]
> showCode((SEval localEnvm1 localEnvm2):ss) state
>  = iConcat[iStr "SEval ",
>           showlocalEnvm1 localEnvm1 state, showlocalEnvm2
>           localEnvm2 state, showCode ss state]
> showCode((Enter adr e1 e2):ss) state
>  = iConcat[iStr "Enter ",iStr "[",iStr(show adr),
>           iStr "] ",showheapnode adr state,showCode ss state]
> showCode((ReturnCon e1 e2):ss) state
>  = iConcat[iStr "ReturnCon ",
>           showCode ss state]
> showCode((ReturnInt e1 e2):ss) state
>  = iConcat[iStr "ReturnInt",showCode ss state]
> showCode [] state = (iStr ".")
> showCode _  state= (iStr "showCode: undefined Code")
```

Die Funktion *showlocalEnvm* zeigt die komplette lokale Umgebung.

```
> showlocalEnvm1 :: LocalEnvm1 -> StgState -> Iseq
> showlocalEnvm1 [] state = iStr ""
> showlocalEnvm1 localEnvm state
```

```
>  = iConcat[iNewline,iStr "where Locals1= ",
>             showlocalEnvm1b localEnvm state]

> showlocalEnvm1b :: LocalEnvm1 -> StgState -> Iseq
> showlocalEnvm1b [] state = iStr ""
> showlocalEnvm1b((name,Integervalue i):ss) state
>  | (null ss)
>     = iConcat[iStr name,iStr " = Int. ",iStr(show i)]
>  | otherwise
>     = iConcat[iStr name,iStr " = Int. ",iStr(show i),
>               iStr "; ",iNewline,showlocalEnvm1b ss state]
> showlocalEnvm1b ((name,Adressvalue a):ss) state
>  | (null ss)
>     = iConcat[iStr name,iStr " = Adr. ",
>               iStr(show a),iStr ": ",a']
>  | otherwise
>     = iConcat[iStr name,iStr " = Adr. ",iStr(show a),
>               iStr": ",a',iStr "; ",iNewline,
>               showlocalEnvm1b ss state]
>    where
>    (HeapItem lambdaform w)=hLookup (getHeap state) a
>    a' = iConcat[iStr" {",showvars fvars,iStr "} ",
>                 showupdate update,
>                 iStr" {",showargs avars,iStr"} -> ",
>                 showInstructions stgexpr]
>    (fvars,update,avars,stgexpr)=lambdaform

> showlocalEnvm2 :: LocalEnvm2 -> StgState -> Iseq
> showlocalEnvm2 [] state = iStr ""
> showlocalEnvm2 localEnvm state
>  = iConcat[iNewline,iStr "where Locals2 = ",
>             showlocalEnvm2b localEnvm state]

> showlocalEnvm2b :: LocalEnvm2 -> StgState -> Iseq
> showlocalEnvm2b [] state = iStr ""
> showlocalEnvm2b ((name,(a,stacks)):ss) state
>  | (null ss)
>     = iConcat[iStr name,iStr " = Adr. ",
>               iStr(show a),iStr ": ",a']
>  | otherwise
>     = iConcat[iStr name,iStr " = Adr. ",
>               iStr(show a),
>               iStr": ",a',iStr "; ",iNewline,
```

```
>               showlocalEnvm2b ss state]
>    where
>    (HeapItem lambdaform w)=hLookup (getHeap state) a
>    a' = iConcat[iStr" {",showvars fvars,iStr "} ",
>                 showupdate update,
>                 iStr" {",showargs avars,iStr"} -> ",
>                 showInstructions stgexpr]
>    (fvars,update,avars,stgexpr)=lambdaform
```

Die Funktion *showvalue* zeigt eine Liste von *Values*, wie sie als Argumente von *ReturnCon* verwendet werden.

```
> showvalues :: [Value] -> Iseq
> showvalues vs = iStr (valuetostring vs)
```

Die Funktion *valuetostring* konvertiert eine Liste von *Values* in einen String.

```
> valuetostring :: [Value] -> [Char]
> valuetostring [] = ""
> valuetostring ((Integervalue i):ss)
>   | (null ss) = "Int:"++(show i)
>   | otherwise = "Int:"++(show i)++";"++(valuetostring ss)
> valuetostring ((Adressvalue a):ss)
>   | (null ss) = "Adr."++(show a)
>   | otherwise = "Adr."++(show a)++";"++(valuetostring ss)
```

Die Funktion *showAs* zeigt den A-Stack. Da der A-Stack von unten nach oben wächst, die Liste, die ihn in Gofer repräsentiert, aber von links nach rechts, also von unten nach oben, muß vor der Ausgabe die Liste durch 'reverse' umgedreht werden.

```
> showAs :: StgState -> Iseq
> showAs state
>  | (null as) = (iStr "A-Stack is empty.")
>  | otherwise
>     = iConcat [(iStr "A-Stack: "),iIndent (iInterleave iNewline
>                (map int_to_iStr (reverse as)))]
```

```
>                       where
>                       as = getAs state

> showBs :: StgState -> Iseq
> showBs state
>  | (null bs) = (iStr "B-Stack is empty.")
>  | otherwise
>     = iConcat [(iStr "B-Stack: "),iIndent (iInterleave iNewline
>                (map int_to_iStr (reverse bs)))]
>                       where
>                       bs = getBs state

> showRegs :: StgState -> Iseq
> showRegs state
>  | (null regs) = (iStr "Return-Registers are emtpy.")
>  | otherwise
>     = iConcat [(iStr "Return-Register: "),
>                iNewline,
>                showatoms regs]
>                       where
>                       regs = getRegs state

> int_to_iStr :: Int -> Iseq
> int_to_iStr i
>  = iStr (show i)

> showStackItem :: StgState -> Value -> Iseq
> showStackItem s (Integervalue i)
>  = iConcat [(iStr "Int: "),iStr (show i)]
> showStackItem s (Adressvalue a)
>  = iConcat [iStr (show a),(iStr ": "),showheapnode a s]

> showheapnode :: Adr -> StgState -> Iseq
> showheapnode adr s
>  = iConcat[iStr "{",showvars fvars,iStr "} ",
>            showupdate update,
>            iStr " {",showargs avars,iStr "}",
>            iStr " -> ",showInstructions stgexpr]
>    where
>    (HeapItem lambdaForm values) = hLookup (getHeap s) adr
>    (fvars,update,avars,stgexpr) = lambdaForm

> showRs state
```

```
>  | (null rs) = iStr "Return-Stack is empty."
>  | otherwise = iConcat[iStr "Return-Stack:",iNewline,showRs2 rs]
>    where
>    rs = getRs state

> showRs2 :: StgRs -> Iseq
> showRs2 [] = iStr ""
> showRs2 ((alts,p1,p2):rs) =
>  iConcat[iStr"(Alts,p1,p2)",iNewline,showRs2 rs]

> showUs state
>  | (null us) = (iStr "Update-Stack is empty.")
>  | otherwise = iConcat[iStr "Update-Stack:",iNewline,showUs2 us]
>    where
>    us = getUs state

> showUs2 :: StgUs -> Iseq
> showUs2 [] = iStr ""
> showUs2 ((as,bs,rs,a):us) =
>  iConcat[iStr"(A-Stack,B-Stack,ReturnStack,Adr.:",
>          iStr(show a),iStr")",iNewline,showUs2 us]

> showStats :: StgState -> Iseq
> showStats s
>  = iNewline
```

A.1.7 Die ran-Funktion

```
> ran :: StgProgram -> [Char]
> ran = showResults . eval . compile
```

A.2 STG-To-G Übersetzer

```
compile :: StgProgram -> GmState
compile program
   = ([],initialCode, [], [], heap, globals, statInitial)
   where (heap, globals) = buildInitialHeap program
```

```
buildInitialHeap :: StgProgram -> (GmHeap, GmGlobals)
buildInitialHeap program
   = mapAccuml allocateSc hInitial compiled
   where compiled = map compileSc program ++
                    compiledPrimitives

type GmCompiledSC = (Name, Int, GmCode)

allocateSc :: GmHeap -> GmCompiledSC -> (GmHeap, (Name, Adr))
allocateSc heap (name, nargs, instns)
      = (heap', (name, adr))
      where (heap', adr) = hAlloc heap (NGlobal nargs instns)

initialCode :: GmCode
initialCode = [Pushglobal "main", Eval, Print]

compileSc :: (Var,LambdaForm) -> GmCompiledSC
compileSc (name,lambdaform)
      = (name,length(args), compileR stgexpr (zip args [0..]))
        where (vars,update,args,stgexpr)=lambdaform

compileR :: GmCompiler
compileR e env = compileE e env ++
                 [Update_ (length(env)),
                  Pop (length(env)), Unwind]

type GmCompiler = StgExpr -> GmEnvironment -> GmCode

type GmEnvironment = Assoc Name Int

compileC :: StgExpr -> GmEnvironment -> GmCode
-- Applikationen
compileC (StgApp v []) env
  | elem v (aDomain env)  = [Push n]
  | otherwise = [Pushglobal v]
    where n = aLookup env v (error "Can't happen")
compileC (StgApp v [(Literal i)]) env
  | elem v (aDomain env) = [Pushint i] ++ [Push n] ++ [Mkap]
  | otherwise = [Pushint i] ++ [Pushglobal v] ++ [Mkap]
    where n = aLookup env v (error "Can't happen")
compileC (StgApp v [(Literal i1),(Literal i2)]) env
  | elem v (aDomain env) = [Pushint i1] ++ [Pushint i2] ++
```

```
    [Push n] ++ [Mkap] ++ [Mkap]
  | otherwise = [Pushint i1] ++ [Pushint i2] ++ [Pushglobal v] ++
    [Mkap] ++ [Mkap]
    where n = aLookup env v (error "Can't happen")
compileC (StgApp v [(Literal i1),(Literal i2),(Literal i3)]) env
  | elem v (aDomain env) = [Pushint i1] ++ [Pushint i2] ++
    [Pushint i3] ++ [Push n] ++ [Mkap] ++ [Mkap]
  | otherwise = [Pushint i1] ++ [Pushint i2] ++ [Pushint i3] ++
    [Pushglobal v] ++ [Mkap] ++ [Mkap]
    where n = aLookup env v (error "Can't happen")
compileC (StgApp v [(Variable v1)]) env
  | ((elem v (aDomain env)) && (elem v1 (aDomain env)))
     -- v und v1 lokal
    = [Push n1] ++ [Push n] ++ [Mkap]
  | elem v (aDomain env) -- v1 ist Global v ist lokal.
    = [Pushglobal v1] ++ [Push n] ++ [Mkap]
  | elem v1 (aDomain env) -- v1 ist lokal, v ist Global.
    = [Push n1] ++ [Pushglobal v] ++ [Mkap]
  | otherwise -- beide sind global.
    = [Pushglobal v1] ++ [Pushglobal v] ++ [Mkap]
    where n = aLookup env v (error "Can't happen")
          n1= aLookup env v1 (error "Can't happen")
compileC (StgApp v [(Literal i1),(Literal i2),(Variable v1)]) env
  | ((elem v (aDomain env)) && (elem v1 (aDomain env)))
     -- v und v1 lokal
    = [Pushint i1] ++ [Pushint i2] ++ [Push n1] ++ [Push n]
      ++ [Mkap] ++ [Mkap] ++ [Mkap]
  | elem v (aDomain env) -- v1 ist Global v ist lokal.
    = [Pushint i1] ++ [Pushint i2] ++ [Pushglobal v1] ++
      [Push n] ++ [Mkap] ++ [Mkap] ++ [Mkap]
  | elem v1 (aDomain env) -- v1 ist lokal, v ist Global.
    = [Pushint i1] ++ [Pushint i2] ++ [Push n1] ++
      [Pushglobal v] ++ [Mkap] ++ [Mkap] ++ [Mkap]
  | otherwise -- beide sind global.
    = [Pushint i1] ++ [Pushint i2] ++ [Pushglobal v1] ++
      [Pushglobal v] ++ [Mkap] ++ [Mkap] ++ [Mkap]
    where n = aLookup env v (error "Can't happen")
          n1= aLookup env v1 (error "Can't happen")
compileC (StgApp v [(Variable v1),(Variable v2)]) env
  | (elem v (aDomain env)) -- v lokal
    = [Push n1] ++ [Push n2] ++ [Push n] ++ [Mkap]
  | otherwise -- v global
    = [Push n1] ++ [Push n2] ++ [Pushglobal v] ++ [Mkap]
```

```
    where n = aLookup env v  (error "Can't happen")
          n1= aLookup env v1 (error "Can't happen")
          n2= aLookup env v2 (error "Can't happen")
compileC (StgApp v [(Variable v1),(Variable v2),(Variable v3)])
         env
  | (elem v (aDomain env)) -- v lokal
    = [Push n1] ++ [Push n2] ++ [Push n3] ++ [Push n] ++ [Mkap]
  | otherwise -- v global
    = [Push n1] ++ [Push n2] ++ [Push n3] ++ [Pushglobal v] ++
      [Mkap]
    where n = aLookup env v  (error "Can't happen")
          n1= aLookup env v1 (error "Can't happen")
          n2= aLookup env v2 (error "Can't happen")
          n3= aLookup env v3 (error "Can't happen")
compileC (StgApp v _) env =
  error "Applikation auf mehr als 3 Arg. nicht implementiert"
compileC (StgLiteral n)    env
 = [Pushint n]
compileC (StgLet binds expr) env
 = compileLet compileC binds expr env
compileC (StgLetrec binds expr) env
 = compileLetrec compileC binds expr env
-- Cons
compileC (StgCons c []) env = [Pack c 0]
compileC (StgCons c [(Literal i)]) env
  = [Pushint i] ++ [Pack c 1]
compileC (StgCons c [(Variable v)]) env
  | (elem v (aDomain env)) -- v ist lokal
    = [Push n] ++ [Pack c 1]
  | otherwise -- v ist global
    = [Pushglobal v] ++ [Pack c 1]
    where
    n = aLookup env v (error "Cannot happen.")
compileC (StgCons c [(Literal i1),(Literal i2)]) env
 = [Pushint i2] ++ [Pushint i1] ++ [Pack c 2]
compileC (StgCons c [(Literal i1),(Variable v2)]) env
 | (elem v2 (aDomain env)) -- v2 ist lokal
   = [Push n2] ++ [Pack c 2]
 | otherwise
   = [Pushglobal v2] ++ [Pushint i1] ++ [Pack c 2]
   where
    n2 = aLookup env v2 (error "Cannot happen.")
compileC (StgCons c [(Variable v1),(Literal i2)]) env
```

```
 = [Pushint i2] ++ [Pushglobal v1] ++ [Pack c 2]
compileC (StgCons c [(Variable v1),(Variable v2)]) env
 | ((elem v1 (aDomain env)) && (elem v2 (aDomain env)))
   -- v1, v2 lokal
   = [Push n2] ++ [Push (n1+1)] ++ [Pack c 2]
 | (elem v1 (aDomain env)) -- v1 lokal, v2 global
   = [Pushglobal v2] ++ [Push (n1+1)] ++ [Pack c 2]
 | (elem v2 (aDomain env)) -- v1 global, v2 lokal
   = [Push n2] ++ [Pushglobal v1] ++ [Pack c 2]
 | otherwise -- v1, v2 global
   = [Pushglobal v2] ++ [Pushglobal v1] ++ [Pack c 2]
   where
    n1 = aLookup env v1 (error "Cannot happen.")
    n2 = aLookup env v2 (error "Cannot happen.")
compileC (StgBiOp biop atoms) env
 = compileBiOp (StgBiOp biop atoms) env

compileE :: StgExpr -> GmEnvironment -> GmCode
compileE (StgLiteral n) env = [Pushint n]
compileE (StgLet binds expr) env
 = compileLet compileE binds expr env
compileE (StgLetrec binds expr) env
 = compileLetrec compileE binds expr env
compileE (StgBiOp biop atoms) env
 = compileBiOp (StgBiOp biop atoms) env
-- case
compileE (StgCase e alts) env
 = compileE e env ++ [Casejump (compileD alts env)]
-- Cons
compileE (StgCons c []) env = [Pack c 0]
compileE (StgCons c [(Literal i)]) env
  = [Pushint i] ++ [Pack c 1]
compileE (StgCons c [(Variable v)]) env
  | (elem v (aDomain env)) -- v ist lokal
    = [Push n] ++ [Pack c 1]
  | otherwise -- v ist global
    = [Pushglobal v] ++ [Pack c 1]
    where
    n = aLookup env v (error "Cannot happen.")
compileE (StgCons c [(Literal i1),(Literal i2)]) env
 = [Pushint i2] ++ [Pushint i1] ++ [Pack c 2]
compileE (StgCons c [(Literal i1),(Variable v2)]) env
  | (elem v2 (aDomain env)) -- v2 ist lokal
```

```
    = [Push n2] ++ [Pushint i1] ++ [Pack c 2]
  | otherwise -- v2 ist global
    = [Pushglobal v2] ++ [Pushint i1] ++ [Pack c 2]
    where
    n2 = aLookup env v2 (error "Cannot happen.")
compileE (StgCons c [(Variable v1),(Literal i2)]) env
  | (elem v1 (aDomain env)) -- v1 ist lokal
    = [Push n1] ++ [Pushint i2] ++ [Pack c 2]
  | otherwise -- v1 ist global
    = [Pushint i2] ++ [Pushglobal v1] ++ [Pack c 2]
    where
    n1 = aLookup env v1 (error "Cannot happen.")
compileE (StgCons c [(Variable v1),(Variable v2)]) env
 | ((elem v1 (aDomain env)) && (elem v2 (aDomain env)))
    -- v1, v2 lokal
   = [Push n2] ++ [Push (n1+1)] ++ [Pack c 2]
 | (elem v1 (aDomain env)) -- v1 lokal, v2 global
   = [Pushglobal v2] ++ [Push (n1+1)] ++ [Pack c 2]
 | (elem v2 (aDomain env)) -- v1 global, v2 lokal
   = [Push n2] ++ [Pushglobal v1] ++ [Pack c 2]
 | otherwise -- v1, v2 global
   = [Pushglobal v2] ++ [Pushglobal v1] ++ [Pack c 2]
   where
    n1 = aLookup env v1 (error "Cannot happen.")
    n2 = aLookup env v2 (error "Cannot happen.")
-- alle sonstigen Expr.
compileE e env = compileC e env ++ [Eval]

bioptogminstr :: Prim -> [Instruction]
-- wird von compileBiOp gebraucht
bioptogminstr biop
           | (biop == Plus)  = [Add_]
           | (biop == Minus) = [Sub_]
           | (biop == Mul)   = [Mul_]
           | (biop == Div)   = [Div_]
           | (biop == Mod)   = [Mod_]
           | (biop == Eq)    = [Eq_]
           | (biop == Lt)    = [Lt_]
           | (biop == Le)    = [Le_]
           | (biop == Gt)    = [Gt_]
           | (biop == Ge)    = [Ge_]
           | (biop == Sqrt)  = [Sqrt_]
           | (biop == Ne)    = [Ne_]
```

```
          | otherwise       = error "undefined operator"

-- Build-in Operations: 4 F"alle
compileBiOp (StgBiOp biop [(Literal i1),(Literal i2)]) env
 = [Pushint i2] ++ [Pushint i1] ++ bioptogminstr biop
compileBiOp (StgBiOp biop [(Variable v1),(Literal i2)]) env
 | (elem v1 (aDomain env))
    = [Pushint i2] ++ [Push (n1+1)] ++ bioptogminstr biop
 | otherwise = error "undeklared variable"
   where
   n1 = aLookup env v1 (error "cannot find variable")
compileBiOp (StgBiOp biop [(Literal i1),(Variable v2)]) env
 | (elem v2 (aDomain env))
    = [Push n2] ++ [Pushint i1] ++ bioptogminstr biop
 | otherwise = error "undeklared variable"
   where
   n2 = aLookup env v2 (error "cannot find variable")
compileBiOp (StgBiOp biop [(Variable v1),(Variable v2)]) env
 | ((elem v2 (aDomain env)) && (elem v1 (aDomain env)))
    = [Push n2] ++ [Push (n1+1)] ++ bioptogminstr biop
 | otherwise = error "undeklared variable"
   where
   n1 = aLookup env v1 (error "cannot find variable")
   n2 = aLookup env v2 (error "cannot find variable")

compileD = compileAlts compileE'

compileAlts comp (Algebraic ([],(DefaultVar var expr))) env
 = [(-999,compileE expr env'++[Slide 1])] --  f"ur default
   where env' = (argOffset 1 env) ++ [(var,0)]
compileAlts comp (Algebraic ([],(DefaultNoVar expr))) env
 = [(-999,compileE expr env)] --  f"ur default
compileAlts comp (Algebraic (((constr,vars,stgexpr):aalts),
                  defaultalt)) env
 = [(constr,comp (length(vars)) stgexpr (zip vars [0..] ++
    argOffset (length(vars)) env))]++compileAlts comp
              (Algebraic (aalts,defaultalt)) env

compileAlts comp (Primitive ([],(DefaultVar var expr))) env
 = [(-999,compileE expr env'++[Slide 1])] --  f"ur default
   where env' = (argOffset 1 env) ++ [(var,0)]
compileAlts comp (Primitive ([],(DefaultNoVar expr))) env
 = [(-999,compileE expr env)] --  f"ur default
```

```
compileAlts comp (Primitive (((lit,stgexpr):palts),
                  defaultalt)) env
 = [(lit,compileE stgexpr env)]++
     compileAlts comp (Primitive (palts,defaultalt)) env
  where vars = []

compileE' offset expr env
 = [Split offset] ++ compileE expr env ++ [Slide offset]

compileLetrec :: GmCompiler -> Binds -> GmCompiler
compileLetrec comp binds expr env
 = [Alloc (length(binds))] ++ compileLetrec' binds env' ++
    comp expr env' ++ [Slide (length(binds))]
   where env' = compileArgs binds env

compileLetrec' :: [(Name,LambdaForm)] -> GmEnvironment -> GmCode
compileLetrec' [] env = []
compileLetrec' ((name,lambdaform):binds) env
 = compileC stgexpr env ++ compileLetrec' binds env
   where (vars,update,args,stgexpr)=lambdaform

compileLet :: GmCompiler -> Binds -> GmCompiler
compileLet comp binds expr env
  = compileLet' binds env ++ comp expr env' ++
                            [Slide (length(binds))]
    where env' = compileArgs binds env

compileArgs :: [(Name,LambdaForm)] -> GmEnvironment
                -> GmEnvironment
compileArgs binds env
     = zip (map fst binds) [n-1, n-2 .. 0] ++ argOffset n env
     where n = length(binds)

compileLet' :: [(Name,LambdaForm)] -> GmEnvironment -> GmCode
compileLet' [] env = []
compileLet' ((name,lambdaform):binds) env
 = compileC stgexpr env ++ compileLet' binds (argOffset 1 env)
   where (vars,update,args,stgexpr)=lambdaform

argOffset :: Int -> GmEnvironment -> GmEnvironment
argOffset n env = [(v, n+m) | (v,m) <- env]

compiledPrimitives :: [GmCompiledSC]
```

```
compiledPrimitives =
[("+",2,[Push 1,Eval,Push 1,Eval,Add_,Update_ 2,Pop 2,Unwind]),
 ("-",2,[Push 1,Eval,Push 1,Eval,Sub_,Update_ 2,Pop 2,Unwind]),
 ("*",2,[Push 1,Eval,Push 1,Eval,Mul_,Update_ 2,Pop 2,Unwind]),
 ("/",2,[Push 1,Eval,Push 1,Eval,Div_,Update_ 2,Pop 2,Unwind]),
 ("mod",2,[Push 1,Eval,Push 1,Eval,Mod_,Update_ 2,Pop 2,Unwind]),
 ("negate",1,[Push 0,Eval,Neg_,Update_ 1,Pop 1,Unwind]),
 ("==",2,[Push 1,Eval,Push 1,Eval,Eq_,Update_ 2,Pop 2,Unwind]),
 ("~=",2,[Push 1,Eval,Push 1,Eval,Ne_,Update_ 2,Pop 2,Unwind]),
 ("<", 2,[Push 1,Eval,Push 1,Eval,Lt_,Update_ 2,Pop 2,Unwind]),
 ("<=",2,[Push 1,Eval,Push 1,Eval,Le_,Update_ 2,Pop 2,Unwind]),
 (">", 2,[Push 1,Eval,Push 1,Eval,Gt_,Update_ 2,Pop 2,Unwind]),
 (">=",2,[Push 1,Eval,Push 1,Eval,Ge_,Update_ 2,Pop 2,Unwind]),
 ("sqrt",2,[Push 1,Eval,Push 1,Eval,Sqrt_,Update_ 2,
  Pop 2,Unwind]),
 ("if",3,[Push 0,Eval,Cond [Push 1] [Push 2], Update_ 3,
  Pop 3, Unwind])]
```

Anhang B

Probeläufe

B.1 STG-Maschinen Interpreter

B.1.1 Fakultätsberechnung

```
Code for main
main = {} \n {} -> fak 3

Code for fak
fak = {} \n {B x} ->
      case - {x,1} of
      0 -> 1
      z -> case fak z of
           res -> * {x,res}
           endcase
      endcase

State transitions

   1) Output://
      Code:
      Eval main .
      A-Stack is empty.
      B-Stack is empty.
      Return-Registers are emtpy.
      Return-Stack is empty.
      Update-Stack is empty.
```

```
2) Output://
   Code:
   Enter [1] {} \n {} -> fak 3.
   A-Stack is empty.
   B-Stack is empty.
   Return-Registers are emtpy.
   Return-Stack is empty.
   Update-Stack is empty.

3) Output://
   Code:
   Eval fak 3.
   A-Stack is empty.
   B-Stack is empty.
   Return-Registers are emtpy.
   Return-Stack is empty.
   Update-Stack is empty.

4) Output://
   Code:
   Enter [2] {} \n {B x} ->
         case - {x,1} of
         0 -> 1
         z -> case fak z of
              res -> * {x,res}.
              endcase
         endcase
   A-Stack is empty.
   B-Stack: 3
   Return-Registers are emtpy.
   Return-Stack is empty.
   Update-Stack is empty.

5) Output://
   Code:
   Eval case - {x,1} of
        0 -> 1
        z -> case fak z of
             res -> * {x,res}
             endcase
        endcase
   where x = Int. 3.
   A-Stack is empty.
```

```
   B-Stack is empty.
   Return-Registers are emtpy.
   Return-Stack is empty.
   Update-Stack is empty.

6) Output://
   Code:
   Eval - {x,1}
   where x = Int. 3.
   A-Stack is empty.
   B-Stack is empty.
   Return-Registers are emtpy.
   Return-Stack:
   (Alts,p1,p2)
   Update-Stack is empty.

7) Output://
   Code:
   ReturnInt.
   A-Stack is empty.
   B-Stack is empty.
   Return-Register:
   2
   Return-Stack:
   (Alts,p1,p2)
   Update-Stack is empty.

8) Output://
   Code:
   Eval case fak z of
        res -> * {x,res}
        endcase
   where x = Int. 3;
         z = Int. 2.
   A-Stack is empty.
   B-Stack is empty.
   Return-Register:
   2
   Return-Stack is empty.
   Update-Stack is empty.

9) Output://
   Code:
```

```
    Eval fak z
    where x = Int. 3;
          z = Int. 2.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    2
    Return-Stack:
    (Alts,p1,p2)
    Update-Stack is empty.

10) Output://
    Code:
    Enter [2] {} \n {B x} ->
          case - {x,1} of
          0 -> 1
          z -> case fak z of
               res -> * {x,res}
               endcase
          endcase
    A-Stack is empty.
    B-Stack: 2
    Return-Register:
    2
    Return-Stack:
    (Alts,p1,p2)
    Update-Stack is empty.

11) Output://
    Code:
    Eval case - {x,1} of
         0 -> 1
         z -> case fak z of
              res -> * {x,res}
              endcase
         endcase
    where x = Int. 2.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    2
    Return-Stack:
    (Alts,p1,p2)
```

```
    Update-Stack is empty.

12) Output://
    Code:
    Eval - {x,1}
    where x = Int. 2.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    2
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

13) Output://
    Code:
    ReturnInt.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    1
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

14) Output://
    Code:
    Eval case fak z of
          res -> * {x,res}
          endcase
    where x = Int. 2;
           z = Int. 1.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    1
    Return-Stack:
    (Alts,p1,p2)
    Update-Stack is empty.

15) Output://
```

```
    Code:
    Eval fak z
    where x = Int. 2;
          z = Int. 1.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    1
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

16) Output://
    Code:
    Enter [2] {} \n {B x} ->
        case - {x,1} of
        0 -> 1
        z -> case fak z of
             res -> * {x,res}
             endcase
        endcase
    A-Stack is empty.
    B-Stack: 1
    Return-Register:
    1
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

17) Output://
    Code:
    Eval case - {x,1} of
         0 -> 1
         z -> case fak z of
              res -> * {x,res}
              endcase
         endcase
    where x = Int. 1.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
```

```
    1
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

18) Output://
    Code:
    Eval - {x,1}
    where x = Int. 1.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    1
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

19) Output://
    Code:
    ReturnInt.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    0
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

20) Output://
    Code:
    Eval 1
    where x = Int. 1.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    0
    Return-Stack:
    (Alts,p1,p2)
```

```
    (Alts,p1,p2)
    Update-Stack is empty.

21) Output://
    Code:
    ReturnInt.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    1
    Return-Stack:
    (Alts,p1,p2)
    (Alts,p1,p2)
    Update-Stack is empty.

22) Output://
    Code:
    Eval * {x,res}
    where x = Int. 2;
          z = Int. 1;
          res = Int. 1.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    1
    Return-Stack:
    (Alts,p1,p2)
    Update-Stack is empty.

23) Output://
    Code:
    ReturnInt.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    2
    Return-Stack:
    (Alts,p1,p2)
    Update-Stack is empty.

24) Output://
    Code:
    Eval * {x,res}
```

```
    where x = Int. 3;
          z = Int. 2;
          res = Int. 2.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    2
    Return-Stack is empty.
    Update-Stack is empty.

25) Output://
    Code:
    ReturnInt.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    6
    Return-Stack is empty.
    Update-Stack is empty.

26) Output:/6/
    Code: no more Code.
    A-Stack is empty.
    B-Stack is empty.
    Return-Register:
    6
    Return-Stack is empty.
    Update-Stack is empty.
```

```
(38628 reductions, 106307 cells, 2 garbage collections)
```

B.2 STG-Code Optimierer

B.2.1 C-Code für STG-Programm ohne Registermanipulationsbefehle

```
#include "runmod.h"
```

```
unsigned *fak_direct(void)
{
label:
if (SpB[-1].val == 1)
{
RetRegU.val = SpB[-2].val;
        SpB -= 2;
{RETURN;}
}
else
{
SpB[-2].val = (SpB[-1].val+SpB[-2].val);
SpB[-1].val = (SpB[-1].val-1);
goto label;
}
}
```

B.2.2 C-Code für STG-Programm mit Registermanipulationsbefehlen

```
#include "runmod.h"

unsigned *fak_core_direct(void)
{
label:
SHOW_STACKS
SHOW_NAME("fak_core ()")
GET_STACKSIZE
if (ar1.val == 1)
{
    RetRegU.val = ar2.val;
    {RETURN;}
}
else
{
ar2.val = ar1.val * ar2.val;
ar1.val = ar1.val - 1;
goto label;
}
}
```

```
unsigned *fak_direct(void)
{
SHOW_STACKS
SHOW_NAME("fak (n#, ak#)")
GET_STACKSIZE
ar1.val = SpB[-1].val;
ar2.val = SpB[-2].val;
/* Arrangiere Stacks */
SpB -= 2;
/* Springe zur globalen Funktion fak_core */
Node = _fak_core;
{JUMP ( fak_core_direct );}
}
```

B.2.3 HP Assembler-Code für STG-Programm ohne Registermanipulationsbefehlen

```
fak_direct:
.PROC
.CALLINFO FRAME=64,NO_CALLS,SAVE_SP,ENTRY_GR=3
.ENTRY
copy %r3,%r1
copy %r30,%r3
stwm %r1,64(0,%r30)
nop
L$0002:
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
ldw 0(0,%r19),%r20
ldo -4(%r20),%r19
ldw 0(0,%r19),%r20
comiclr,= 1,%r20,0
bl,n L$0003,0
addil L'RetRegU-$global$,%r27
ldo R'RetRegU-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -8(%r21),%r20
```

```
ldw 0(0,%r20),%r21
stw %r21,0(0,%r19)
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -8(%r21),%r20
stw %r20,0(0,%r19)
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -4(%r21),%r20
copy %r20,%r21
stw %r21,0(0,%r19)
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r19
ldw 0(0,%r19),%r28
bl,n L$0001,0
bl,n L$0004,0
L$0003:
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
ldw 0(0,%r19),%r20
ldo -8(%r20),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -4(%r21),%r20
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r21
ldw 0(0,%r21),%r22
ldo -8(%r22),%r21
ldw 0(0,%r20),%r20
ldw 0(0,%r21),%r21
add %r20,%r21,%r20
stw %r20,0(0,%r19)
addil L'SpB-$global$,%r27
```

```
ldo R'SpB-$global$(%r1),%r19
ldw 0(0,%r19),%r20
ldo -4(%r20),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -4(%r21),%r20
ldw 0(0,%r20),%r21
ldo -1(%r21),%r20
stw %r20,0(0,%r19)
bl,n L$0002,0
L$0004:
L$0001:
ldo 64(%r3),%r30
ldwm -64(0,%r30),%r3
bv,n 0(%r2)
.EXIT
.PROCEND
.SPACE $PRIVATE$
.SUBSPA $BSS$
```

B.2.4 HP Assembler-Code für STG-Programm mit Registermanipulationsbefehlen

```
fak_core_direct:
.PROC
.CALLINFO FRAME=64,NO_CALLS,SAVE_SP,ENTRY_GR=3
.ENTRY
copy %r3,%r1
copy %r30,%r3
stwm %r1,64(0,%r30)
nop
L$0002:
addil L'ar1-$global$,%r27
ldo R'ar1-$global$(%r1),%r19
ldw 0(0,%r19),%r20
comiclr,= 1,%r20,0
bl,n L$0003,0
addil L'RetRegU-$global$,%r27
ldo R'RetRegU-$global$(%r1),%r19
```

```
addil L'ar2-$global$,%r27
ldo R'ar2-$global$(%r1),%r20
ldw 0(0,%r20),%r21
stw %r21,0(0,%r19)
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -4(%r21),%r20
copy %r20,%r21
stw %r21,0(0,%r19)
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r19
ldw 0(0,%r19),%r28
bl,n L$0001,0
bl,n L$0004,0
L$0003:
addil L'ar2-$global$,%r27
ldo R'ar2-$global$(%r1),%r19
addil L'ar1-$global$,%r27
ldo R'ar1-$global$(%r1),%r20
addil L'ar2-$global$,%r27
ldo R'ar2-$global$(%r1),%r21
fldws 0(0,%r20),%fr8
fldws 0(0,%r21),%fr9
xmpyu %fr8,%fr9,%fr8
fstws %fr8R,0(0,%r19)
addil L'ar1-$global$,%r27
ldo R'ar1-$global$(%r1),%r19
addil L'ar1-$global$,%r27
ldo R'ar1-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -1(%r21),%r20
stw %r20,0(0,%r19)
bl,n L$0002,0
L$0004:
L$0001:
ldo 64(%r3),%r30
ldwm -64(0,%r30),%r3
```

```
bv,n 0(%r2)
.EXIT
.PROCEND
.align 4
.EXPORT fak_direct,CODE
.EXPORT fak_direct,ENTRY,PRIV_LEV=3,RTNVAL=GR
fak_direct:
.PROC
.CALLINFO FRAME=64,NO_CALLS,SAVE_SP,ENTRY_GR=3
.ENTRY
copy %r3,%r1
copy %r30,%r3
stwm %r1,64(0,%r30)
addil L'ar1-$global$,%r27
ldo R'ar1-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -4(%r21),%r20
ldw 0(0,%r20),%r21
stw %r21,0(0,%r19)
addil L'ar2-$global$,%r27
ldo R'ar2-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -8(%r21),%r20
ldw 0(0,%r20),%r21
stw %r21,0(0,%r19)
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r19
addil L'SpB-$global$,%r27
ldo R'SpB-$global$(%r1),%r20
ldw 0(0,%r20),%r21
ldo -8(%r21),%r20
stw %r20,0(0,%r19)
ldil LP'fak_core_direct,%r19
ldo RP'fak_core_direct(%r19),%r28
bl,n L$0005,0
L$0005:
ldo 64(%r3),%r30
ldwm -64(0,%r30),%r3
bv,n 0(%r2)
```

```
.EXIT
.PROCEND
.SPACE $PRIVATE$
.SUBSPA $BSS$
```

B.3 STG-To-G Übersetzer

Wir verwenden zunächst den STG-To-G Übersetzer, um das STG-Programm zur Fakultätsberechnung[1] in G-Code zu übersetzen. Dann lassen wir das G-Programm durch die modifizierte G-Maschine, die auch die Updateinformation im STG-Programm berücksichtigt, ablaufen.

Wir können damit die Auswertung des gleichen funktionalen Programms durch die STG-Maschine und die G-Maschine vergleichen.

B.3.1 Übersetzung des STG-Fakultätsprogramms in G-Code

```
Code for fak

Pushint 1
Push 1
Sub
Casejump 0      : Pushint 1
Casejump default: Push 0
                  Pushglobal fak
                  Mkap
                  Eval
                  Casejump default: Push 0
                                    Push 3
                                    Mul
                                    Slide 1
                  Slide 1
Unwind

Code for main

Pushint 3
```

[1] Bereits angegeben beim Probelauf des STG-Maschinen Interpreters.

```
Pushglobal fak
Mkap
Eval
Unwind
```

B.3.2 Ausführung des nach G-Code übersetzten Fakultätsprogramms durch die modizifierte G-Maschine

```
State transitions

   1) Output://
       Stack:[]
       Dump:[]
        Code:{Pushglobal main
              Eval
              Print}


   2) Output://
       Stack:[2: Global main]
       Dump:[]
        Code:{Eval
              Print}


   3) Output://
       Stack:[2: Global main]
       Dump:[<{Print}, []>]
        Code:{Unwind}


   4) Output://
       Stack:[2: Global main]
       Dump:[<{Print}, []>]
        Code:{Pushint 3
              Pushglobal fak
              Mkap
              Eval
              Unwind}
```

```
5) Output://
    Stack:[2: Global main
           17: 3]
    Dump:[<{Print}, []>]
     Code:{Pushglobal fak
           Mkap
           Eval
           Unwind}


6) Output://
    Stack:[2: Global main
           17: 3
           1: Global fak]
    Dump:[<{Print}, []>]
     Code:{Mkap
           Eval
           Unwind}


7) Output://
    Stack:[2: Global main
           18: Ap 1 17]
    Dump:[<{Print}, []>]
     Code:{Eval
           Unwind}


8) Output://
    Stack:[18: Ap 1 17]
    Dump:[<{Print}, []>
          <{Unwind}, [2]>]
     Code:{Unwind}


9) Output://
    Stack:[18: Ap 1 17
           1: Global fak]
    Dump:[<{Print}, []>
          <{Unwind}, [2]>]
     Code:{Unwind}
```

```
10) Output://
     Stack:[18: Ap 1 17
            17: 3]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Pushint 1
            Push 1
            Sub
            Casejump 0  Code:{Pushint 1}
            default  Code:{Push 0
                           Pushglobal fak
                           Mkap
                           Eval
                           Casejump default  Code:{Push 0
                                                   Push 3
                                                   Mul
                                                   Slide 1}

                           Slide 1}

            Unwind}


11) Output://
     Stack:[18: Ap 1 17
            17: 3
            19: 1]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Push 1
            Sub
            Casejump 0  Code:{Pushint 1}
            default  Code:{Push 0
                           Pushglobal fak
                           Mkap
                           Eval
                           Casejump default  Code:{Push 0
                                                   Push 3
                                                   Mul
                                                   Slide 1}
```

```
                       Slide 1}

           Unwind}


12) Output://
     Stack:[18: Ap 1 17
            17: 3
            19: 1
            17: 3]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Sub
            Casejump 0  Code:{Pushint 1}
            default  Code:{Push 0
                       Pushglobal fak
                       Mkap
                       Eval
                       Casejump default  Code:{Push 0
                                               Push 3
                                               Mul
                                               Slide 1}

                       Slide 1}

           Unwind}


13) Output://
     Stack:[18: Ap 1 17
            17: 3
            20: 2]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Casejump 0  Code:{Pushint 1}
            default  Code:{Push 0
                       Pushglobal fak
                       Mkap
                       Eval
                       Casejump default  Code:{Push 0
                                               Push 3
                                               Mul
                                               Slide 1}
```

```
                              Slide 1}

              Unwind}


  14) Output://
        Stack:[18: Ap 1 17
               17: 3
               20: 2]
        Dump:[<{Print}, []>
              <{Unwind}, [2]>]
         Code:{Push 0
               Pushglobal fak
               Mkap
               Eval
               Casejump default  Code:{Push 0
                                       Push 3
                                       Mul
                                       Slide 1}

               Slide 1
               Unwind}


  15) Output://
        Stack:[18: Ap 1 17
               17: 3
               20: 2
               20: 2]
        Dump:[<{Print}, []>
              <{Unwind}, [2]>]
         Code:{Pushglobal fak
               Mkap
               Eval
               Casejump default  Code:{Push 0
                                       Push 3
                                       Mul
                                       Slide 1}

               Slide 1
               Unwind}
```

```
16) Output://
     Stack:[18: Ap 1 17
            17: 3
            20: 2
            20: 2
            1: Global fak]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Mkap
            Eval
            Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}

            Slide 1
            Unwind}


17) Output://
     Stack:[18: Ap 1 17
            17: 3
            20: 2
            21: Ap 1 20]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Eval
            Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}

            Slide 1
            Unwind}


18) Output://
     Stack:[21: Ap 1 20]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
```

```
                                     Mul
                                     Slide 1}
          ; Slide 1; Unwind}, [20, 17, 18]>]
     Code:{Unwind}


19) Output://
    Stack:[21: Ap 1 20
           1: Global fak]
    Dump:[<{Print}, []>
          <{Unwind}, [2]>
          <{Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}
          ; Slide 1; Unwind}, [20, 17, 18]>]
     Code:{Unwind}


20) Output://
    Stack:[21: Ap 1 20
           20: 2]
    Dump:[<{Print}, []>
          <{Unwind}, [2]>
          <{Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}
          ; Slide 1; Unwind}, [20, 17, 18]>]
     Code:{Pushint 1
           Push 1
           Sub
           Casejump 0  Code:{Pushint 1}
           default  Code:{Push 0
                          Pushglobal fak
                          Mkap
                          Eval
                          Casejump default  Code:{Push 0
                                                  Push 3
                                                  Mul
                                                  Slide 1}

                          Slide 1}
```

```
             Unwind}


 21) Output://
          Stack:[21: Ap 1 20
                 20: 2
                 19: 1]
          Dump:[<{Print}, []>
                <{Unwind}, [2]>
                <{Casejump default  Code:{Push 0
                                         Push 3
                                         Mul
                                         Slide 1}
                ; Slide 1; Unwind}, [20, 17, 18]>]
           Code:{Push 1
                 Sub
                 Casejump 0  Code:{Pushint 1}
                 default  Code:{Push 0
                               Pushglobal fak
                               Mkap
                               Eval
                               Casejump default  Code:{Push 0
                                                       Push 3
                                                       Mul
                                                       Slide 1}

                               Slide 1}

                 Unwind}


 22) Output://
          Stack:[21: Ap 1 20
                 20: 2
                 19: 1
                 20: 2]
          Dump:[<{Print}, []>
                <{Unwind}, [2]>
                <{Casejump default  Code:{Push 0
                                         Push 3
                                         Mul
                                         Slide 1}
```

```
           ; Slide 1; Unwind}, [20, 17, 18]>]
      Code:{Sub
            Casejump 0  Code:{Pushint 1}
            default  Code:{Push 0
                           Pushglobal fak
                           Mkap
                           Eval
                           Casejump default  Code:{Push 0
                                                   Push 3
                                                   Mul
                                                   Slide 1}

                           Slide 1}

            Unwind}


 23) Output://
     Stack:[21: Ap 1 20
            20: 2
            22: 1]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>]
      Code:{Casejump 0  Code:{Pushint 1}
            default  Code:{Push 0
                           Pushglobal fak
                           Mkap
                           Eval
                           Casejump default  Code:{Push 0
                                                   Push 3
                                                   Mul
                                                   Slide 1}

                           Slide 1}

            Unwind}
```

```
24) Output://
     Stack:[21: Ap 1 20
            20: 2
            22: 1]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>]
      Code:{Push 0
            Pushglobal fak
            Mkap
            Eval
            Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}

            Slide 1
            Unwind}


25) Output://
     Stack:[21: Ap 1 20
            20: 2
            22: 1
            22: 1]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>]
      Code:{Pushglobal fak
            Mkap
            Eval
            Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}
```

```
            Slide 1
            Unwind}


 26) Output://
     Stack:[21: Ap 1 20
            20: 2
            22: 1
            22: 1
            1: Global fak]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>]
      Code:{Mkap
            Eval
            Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}

            Slide 1
            Unwind}


 27) Output://
     Stack:[21: Ap 1 20
            20: 2
            22: 1
            23: Ap 1 22]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>]
      Code:{Eval
            Casejump default  Code:{Push 0
```

```
                                           Push 3
                                           Mul
                                           Slide 1}

                 Slide 1
                 Unwind}


28) Output://
    Stack:[23: Ap 1 22]
    Dump:[<{Print}, []>
          <{Unwind}, [2]>
          <{Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}
          ; Slide 1; Unwind}, [20, 17, 18]>
          <{Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}
          ; Slide 1; Unwind}, [22, 20, 21]>]
     Code:{Unwind}


29) Output://
    Stack:[23: Ap 1 22
           1: Global fak]
    Dump:[<{Print}, []>
          <{Unwind}, [2]>
          <{Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}
          ; Slide 1; Unwind}, [20, 17, 18]>
          <{Casejump default  Code:{Push 0
                                    Push 3
                                    Mul
                                    Slide 1}
          ; Slide 1; Unwind}, [22, 20, 21]>]
     Code:{Unwind}
```

```
30) Output://
     Stack:[23: Ap 1 22
            22: 1]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [22, 20, 21]>]
      Code:{Pushint 1
            Push 1
            Sub
            Casejump 0  Code:{Pushint 1}
            default  Code:{Push 0
                           Pushglobal fak
                           Mkap
                           Eval
                           Casejump default  Code:{Push 0
                                                   Push 3
                                                   Mul
                                                   Slide 1}

                           Slide 1}

            Unwind}


31) Output://
     Stack:[23: Ap 1 22
            22: 1
            19: 1]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
```

```
            ; Slide 1; Unwind}, [20, 17, 18]>
            <{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}
            ; Slide 1; Unwind}, [22, 20, 21]>]
       Code:{Push 1
             Sub
             Casejump 0  Code:{Pushint 1}
             default  Code:{Push 0
                            Pushglobal fak
                            Mkap
                            Eval
                            Casejump default  Code:{Push 0
                                                    Push 3
                                                    Mul
                                                    Slide 1}

                            Slide 1}

             Unwind}


  32) Output://
      Stack:[23: Ap 1 22
             22: 1
             19: 1
             22: 1]
      Dump:[<{Print}, []>
            <{Unwind}, [2]>
            <{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}
            ; Slide 1; Unwind}, [20, 17, 18]>
            <{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}
            ; Slide 1; Unwind}, [22, 20, 21]>]
       Code:{Sub
             Casejump 0  Code:{Pushint 1}
             default  Code:{Push 0
```

```
                              Pushglobal fak
                              Mkap
                              Eval
                              Casejump default  Code:{Push 0
                                                      Push 3
                                                      Mul
                                                      Slide 1}

                              Slide 1}

               Unwind}


 33) Output://
        Stack:[23: Ap 1 22
               22: 1
               24: 0]
        Dump:[<{Print}, []>
              <{Unwind}, [2]>
              <{Casejump default  Code:{Push 0
                                        Push 3
                                        Mul
                                        Slide 1}
              ; Slide 1; Unwind}, [20, 17, 18]>
              <{Casejump default  Code:{Push 0
                                        Push 3
                                        Mul
                                        Slide 1}
              ; Slide 1; Unwind}, [22, 20, 21]>]
         Code:{Casejump 0  Code:{Pushint 1}
               default  Code:{Push 0
                              Pushglobal fak
                              Mkap
                              Eval
                              Casejump default  Code:{Push 0
                                                      Push 3
                                                      Mul
                                                      Slide 1}

                              Slide 1}

               Unwind}
```

```
34) Output://
     Stack:[23: Ap 1 22
            22: 1
            24: 0]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [22, 20, 21]>]
      Code:{Pushint 1
            Unwind}


35) Output://
     Stack:[23: Ap 1 22
            22: 1
            24: 0
            19: 1]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [20, 17, 18]>
           <{Casejump default  Code:{Push 0
                                     Push 3
                                     Mul
                                     Slide 1}
           ; Slide 1; Unwind}, [22, 20, 21]>]
      Code:{Unwind}


36) Output://
     Stack:[21: Ap 1 20
```

```
              20: 2
              22: 1
              19: 1]
       Dump:[<{Print}, []>
             <{Unwind}, [2]>
             <{Casejump default  Code:{Push 0
                                       Push 3
                                       Mul
                                       Slide 1}
             ; Slide 1; Unwind}, [20, 17, 18]>]
        Code:{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}

              Slide 1
              Unwind}


  37) Output://
       Stack:[21: Ap 1 20
              20: 2
              22: 1
              19: 1]
       Dump:[<{Print}, []>
             <{Unwind}, [2]>
             <{Casejump default  Code:{Push 0
                                       Push 3
                                       Mul
                                       Slide 1}
             ; Slide 1; Unwind}, [20, 17, 18]>]
        Code:{Push 0
              Push 3
              Mul
              Slide 1
              Slide 1
              Unwind}


  38) Output://
       Stack:[21: Ap 1 20
              20: 2
              22: 1
```

```
        19: 1
        19: 1]
 Dump:[<{Print}, []>
       <{Unwind}, [2]>
       <{Casejump default  Code:{Push 0
                                 Push 3
                                 Mul
                                 Slide 1}
       ; Slide 1; Unwind}, [20, 17, 18]>]
  Code:{Push 3
        Mul
        Slide 1
        Slide 1
        Unwind}


39) Output://
 Stack:[21: Ap 1 20
        20: 2
        22: 1
        19: 1
        19: 1
        20: 2]
 Dump:[<{Print}, []>
       <{Unwind}, [2]>
       <{Casejump default  Code:{Push 0
                                 Push 3
                                 Mul
                                 Slide 1}
       ; Slide 1; Unwind}, [20, 17, 18]>]
  Code:{Mul
        Slide 1
        Slide 1
        Unwind}


40) Output://
 Stack:[21: Ap 1 20
        20: 2
        22: 1
        19: 1
        25: 2]
 Dump:[<{Print}, []>
```

```
            <{Unwind}, [2]>
            <{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}
            ; Slide 1; Unwind}, [20, 17, 18]>]
       Code:{Slide 1
             Slide 1
             Unwind}


 41) Output://
      Stack:[21: Ap 1 20
             20: 2
             22: 1
             25: 2]
      Dump:[<{Print}, []>
            <{Unwind}, [2]>
            <{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}
            ; Slide 1; Unwind}, [20, 17, 18]>]
       Code:{Slide 1
             Unwind}


 42) Output://
      Stack:[21: Ap 1 20
             20: 2
             25: 2]
      Dump:[<{Print}, []>
            <{Unwind}, [2]>
            <{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}
            ; Slide 1; Unwind}, [20, 17, 18]>]
       Code:{Unwind}


 43) Output://
      Stack:[18: Ap 1 17
```

```
              17: 3
              20: 2
              25: 2]
       Dump:[<{Print}, []>
             <{Unwind}, [2]>]
        Code:{Casejump default  Code:{Push 0
                                      Push 3
                                      Mul
                                      Slide 1}

              Slide 1
              Unwind}


  44) Output://
       Stack:[18: Ap 1 17
              17: 3
              20: 2
              25: 2]
       Dump:[<{Print}, []>
             <{Unwind}, [2]>]
        Code:{Push 0
              Push 3
              Mul
              Slide 1
              Slide 1
              Unwind}


  45) Output://
       Stack:[18: Ap 1 17
              17: 3
              20: 2
              25: 2
              25: 2]
       Dump:[<{Print}, []>
             <{Unwind}, [2]>]
        Code:{Push 3
              Mul
              Slide 1
              Slide 1
              Unwind}
```

```
46) Output://
     Stack:[18: Ap 1 17
            17: 3
            20: 2
            25: 2
            25: 2
            17: 3]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Mul
            Slide 1
            Slide 1
            Unwind}


47) Output://
     Stack:[18: Ap 1 17
            17: 3
            20: 2
            25: 2
            26: 6]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Slide 1
            Slide 1
            Unwind}


48) Output://
     Stack:[18: Ap 1 17
            17: 3
            20: 2
            26: 6]
     Dump:[<{Print}, []>
           <{Unwind}, [2]>]
      Code:{Slide 1
            Unwind}


49) Output://
     Stack:[18: Ap 1 17
            17: 3
```

```
              26: 6]
       Dump:[<{Print}, []>
             <{Unwind}, [2]>]
        Code:{Unwind}

  50) Output://
       Stack:[2: Global main
              26: 6]
       Dump:[<{Print}, []>]
        Code:{Unwind}

  51) Output://
       Stack:[26: 6]
       Dump:[]
        Code:{Print}

  52) Output:/6/
       Stack:[]
       Dump:[]
        Code:{}

Steps taken = 51
(118423 reductions, 326134 cells, 5 garbage collections)
```

Literaturverzeichnis

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill, New York, 1985.

[Aug84] Lennart Augustsson. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, 1984.

[Cun94] H. Conrad Cunningham. Notes on functional programming with gofer. Technical Report UMCIS-1994-01, Dept of Computer and Information Science, University of Mississippi, January 1994.

[Dav92] A.J.T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, Cambridge, 1992.

[FW87] Fairbairn and Wray. *TIM - a simple lazy abstract machine to execute supercombinators.* Functional Programming Languages and Computer Architecture, LNCS 274. Springer Verlag, 1987.

[Hew91] Hewlett Packard. *HP 9000 Computer Systems Assembly Language Reference Manual*, fourth edition, January 1991.

[Hin92] Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda.* Teubner, Stuttgart, 1992.

[Hol91] Ian Holyer. *Functional Programming with Miranda.* Pitman, 1991.

[HPW+92] Paul Hudak [ed.], Simon L. Peyton Jones [ed.], Philip Wadler [ed.], Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. A non-strict purely functional language. version 1.2, 1992.

[Jon91] Mark P Jones. An introduction to gofer, 1991.

[Jon94] Mark P. Jones. The implementation of the gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.

[Kie95] Frank Kiesel. Implementierung eines Übersetzers zur Erzeugung effizientem C-Code aus funktionalen STG-Code. Master's thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1995. in German.

[NSvP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Ekelen, and M. J. Plasmeijer. Concurrent Clean. In Springer Verlag, editor, *Proc of Parallel Architecture and Languages Europe (PARLE'91)*, number 505 in Lecture Notes in Computer Science, pages 202–219, 1991.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, London, 1987.

[PJ92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[PJHH+92] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1992.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall International, London, 1991.

[Sch94] Marko Schütz. Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache. Master's thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1994. in German.

[Ste90] Guy L. Steele. *Common Lisp*. Digital Press, 1990.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer, 1985.