# Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache

Marko Schütz

**Diplomarbeit**
1. August 1994

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-Erik Panitz und Prof. Dr. Manfred Schmidt-Schauß für ihre ausgezeichnete Betreuung und ihre wertvollen Anregungen.

Marko Schütz

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 1. August 1994

Marko Schütz

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Nicht-strikt auswertende funktionale Programmiersprachen gewinnen zunehmend an Bedeutung. Dafür gibt es viele Gründe. Einer ist sicherlich die Möglichkeit der automatischen Parallelisierung funktionaler Programme. Um dabei jedoch eine nicht-strikte Semantik zu erhalten, möchte man zwischen Teilausdrücken, die ausgewertet werden *müssen* und solchen, die ausgewertet werden *können*, unterscheiden. Diese Information kann Striktheits-Analyse liefern. Es zeigt sich, daß diese Information auch bei der Erzeugung effizienten Codes für sequentielle Maschinen unerläßlich ist und bestimmte Optimierungen überhaupt erst möglich macht.

Seit dem Ende der '70er Jahre werden verschiedene Programmanalysen mittels abstrakter Interpretation untersucht. Ein wesentlicher Nachteil der abstrakten Interpretation ist die Notwendigkeit, Fixpunkte von Funktionen zu berechnen, was in der Praxis effiziente Implementationen verhindert hat.

Auf Ideen der abstrakten Interpretation aufbauend hat Nöcker in [Nö93] die Methode der abstrakten Reduktion entwickelt, die im Gegensatz zur abstrakten Interpretation effizient umzusetzen ist und mit der viel Striktheits-Information gefunden wird.

Die bisher einzige Implementation seiner Methode wurde von Nöcker selbst für Concurrent Clean in C vorgenommen.

In der vorliegenden Arbeit entwerfen wir ein Maschinenmodell, die $G^{\#}$-Maschine, das wir in der funktionalen Programmiersprache Haskell implementieren.

## 1.1 Überblick

In Kapitel 2 fassen wir einige wesentliche Eigenschaften funktionaler Programmiersprachen zusammen. Pattern-Matching spielt hier eine zentrale Rolle, unter anderem wegen des Zusammenhangs zwischen Pattern-Matching und Striktheits-Analyse. Wir beschreiben kurz Haskell bzw. Gofer, die Sprache, in der die Implementation zu dieser Arbeit programmiert wurde. Weiter beschreiben wir Concurrent Clean, eine funktionale Sprache, die das Umfeld ist, in dem die Methoden, die dieser Arbeit zugrundeliegen, entstanden sind. Wir fahren fort, den Sprachkern zu definieren, der eine minimale funktionale Sprache darstellt, eine Zwischensprache für höhere funktionale Sprachen wie Haskell oder Concurrent Clean. Der Auswertung funktionaler Programme liegt häufig eine Term- bzw. Graphersetzungssemantik zugrunde. In Abschnitt 2.3 führen wir die G-Maschine ein, ein Maschinenmodell, das Graphreduktion effizient umsetzt. Die Graphersetzungen bei der Auswertung funktionaler Programme können mit der Information, die die Striktheits-Analyse liefert, optimiert werden. Schließlich beschreiben wir, wie die Information zur Optimierung des erzeugten Codes eingesetzt werden kann.

Kapitel 3 erklärt abstrakte Reduktion, die verwendete Methode, die Mengen von Ausdrücken in Relation zueinander setzt. Dabei werden konkrete Reduktionen nachgebildet. Um Striktheit zu analysieren, wird versucht, durch Anwendung abstrakter Reduktionen den minimalen Ausdruck abzuleiten. Es wird die Ordnung definiert, bzgl. der dieser Ausdruck minimal ist. Wir definieren einfache Vereinigungen, die uns später die Terminierung des Pattern-Matching liefern. Wir zeigen, wie Pattern-Matching für abstrakte Werte arbeitet und wie die o. g. Ordnung approximiert werden kann. Schließlich erklären wir Zyklus- und $\perp$-Einführung, zwei Methoden, bei denen durch Untersuchung des Reduktionspfades abstrakte Reduktionen hergeleitet werden.

In Kapitel 4 gehen wir auf die Umsetzung der in Kapitel 3 beschriebenen Methoden ein. Ein wesentliches Merkmal unserer Umsetzung ist die Verwendung eines neuen Maschinenmodells, der $G^{\#}$-Maschine. Zunächst erklären wir in welchen Teilen Änderungen gegenüber der G-Maschine notwendig sind, um abstrakte Graphersetzung umzusetzen, um anschließend zu beschreiben, welche Bereiche der $G^{\#}$-Maschine die Pfadanalyse umsetzen und in welcher Weise dies geschieht. Im Besonderen argumentieren wir, daß auf strikte Auswertung, die in [Nö93] für notwendig gehalten wird, verzichtet werden kann. In 4.2 zeigen wir, wie wenig Aufwand nötig ist, um Notwendigkeit der durchgeführten Reduktionen in unserer Umsetzung sicherzustellen. Anschließend betrachten wir das Speichern und Verwenden bereits ermittelter Striktheits-Information. Beides kann durch Änderung bzw. Ausführung von $G^{\#}$-Code geschehen, was den Vorteil hat, daß die $G^{\#}$-Maschine vom Konzept der Striktheit bzw. der Striktheits-Information völlig unabhängig ist. Schließlich definieren wir die $G^{\#}$-Maschine durch ihre operationale Semantik.

Im Kapitel 5 betrachten wir zwei alternative Methoden zur Ermittlung von Striktheit: das Propagieren strikter Kontexte und Striktheits-Analyse mittels abstrakter Interpretation. In Abschnitt 5.3 vergleichen wir eine Implementation der abstrakten Interpretation mit unserer Implementation der abstrakten Reduktion und betrachten Funktionen, für die die erstere keine Striktheits-Information findet, letztere jedoch schon.

Kapitel 6 diskutiert die Resultate der Analyse von Programmteilen der Striktheits-Analyse und wir fassen dort die vorliegende Arbeit zusammen. In Abschnitt 6.2 zeigen wir, in welchen Bereichen weiterführende Untersuchungen wünschenswert wären.

# Kapitel 2

# Grundlagen

## 2.1 Funktionale Sprachen

Funktionale Programmiersprachen zeichnen sich durch wesentliche Eigenschaften und Konzepte aus, die sie von den imperativen Sprachen unterscheiden.

So ist z. B. ein wesentlicher Nachteil von „von Neumann-Sprachen“, wie sie Backus in [Bac78] nennt, daß die implizite Veränderung, die Konstrukte mit Seiteneffekten bewirken, ein beliebiges Umordnen der Konstrukte unmöglich macht. Programme sind dann abhängig von der Reihenfolge, in der ihre Konstrukte ausgewertet werden. Sogar Konstrukte, die zunächst in keinem Abhängigkeitsverhältnis zueinander stehen müssen vom Programmierer aufeinanderfolgend angeordnet werden und erhalten so eine vorgegebene Auswertungsreihenfolge. Darüberhinaus gibt es Konstrukte, die verhindern, daß mit konventioneller Mathematik Schlüsse über solche Programme gefolgert werden können. Dies führt dazu, daß Aussagen über Programme in „von Neumann-Sprachen“ ungleich schwieriger herzuleiten sind als entsprechende Aussagen in funktionalen Programmiersprachen. Das zeigen unter anderem die Arbeiten von Baber [Bab87], Gries [Gri81] und Dijkstra [Dij76]. In der dort vorgeschlagenen Methode ist es immer notwendig, die Bedeutung eines Ausdrucks durch die Auswirkung, die seine Auswertung auf den Zustand eines Programms hat, zu definieren. Der Zustand wird dargestellt durch eine Datenumgebung, die jede Variable im Programm an einen Wert bindet.

In funktionalen Sprachen gibt es keine Seiteneffekte, daher ist die Auswertung der Teilausdrücke in jeder beliebigen Reihenfolge, einschließlich parallel, möglich. Eine Sprache, in der der Wert eines Ausdrucks nur von den Werten seiner Teilausdrücke abhängt, nennt man *referentiell transparent*. Bei referentieller Transparenz genügt uns als Bedeutung eines Ausdrucks dessen Wert und wir können es als die Aufgabe des Computers ansehen, diesen zu bestimmen. Zusätzlich zu der Unabhängigkeit von der Auswertungsreihenfolge ermöglicht referentielle Transparenz die Benutzung modularer Beweistechniken und ganz allgemein die Ausdrücke in funktionalen Sprachen zu behandeln, wie jede andere Art mathematischer Ausdrücke, was die Konstruktion, Manipulation und Beweisführung angeht.

Funktionale Programmiersprachen werden (zur Zeit noch) hauptsächlich für das schnelle Erstellen von Prototypen verwendet. Dazu sind sie besonders geeignet, denn sie erlauben die grundlegenden Aspekte des Prototyps darzustellen, ohne sich mit zu vielen Details aufzuhalten. Die resultierenden Programme sind sehr kurz im Vergleich zu ihren imperativen Entsprechungen. So kurz, daß Davie in [Dav92] vor der „Einzeiler-Krankheit" warnt, die Programmierer bestimmter Sprachen, wie APL, häufig zu befallen scheint.

Woraus also bestehen funktionale Programme? Sie bestehen aus zwei Teilen, einer Reihe von Funktionsdefinitionen und einem Ausdruck, der ausgewertet werden soll. Ausgewertet wird dieser Ausdruck, indem er in seine einfachste äquivalente Form gebracht wird und diese dann ausgegeben wird. Die Reduktionsschritte kann man dazu durch Term- bzw. Graphersetzung beschreiben. Es wird ein Teilausdruck, in dem eine Funktion angewandt wird, durch die rechte Seite der Funktionsdefinition ersetzt, wobei die formalen Parameter der Funktion durch die tatsächlichen Parameter ersetzt werden. Hat man z. B. `f` als

```
f x y = x + y
```

definiert, so kann folgendermaßen reduziert werden:

$$\texttt{f 1 2} \to \texttt{1 + 2} \to \texttt{3}.$$

Für diese Ersetzungen und Vereinfachungen werden sowohl primitive als auch vom Programmierer, in Form von Definitionen, gegebene Regeln verwendet.

Ein Ausdruck, der nicht weiter reduziert werden kann, ist *kanonisch* oder in *Normalform*. Es gibt allerdings Werte, die keine kanonische Darstellung haben, andere haben keine endliche, wie die Zahl $\pi$. Bezeichnet der `/`-Operator die primitive Division, dann stellt `1/0` keinen wohl-*definierten* Wert dar. Damit nun jedem wohl-*geformten* Ausdruck ein Wert zugeordnet werden kann, führt man ein spezielles Symbol $\perp$ ein, das für den undefinierten Wert steht. Damit wird der Wert von `1/0` zu $\perp$, aber man verlangt nicht, daß der Computer in der Lage ist, den Wert $\perp$ darzustellen. $\perp$ steht auch für Ausdrücke, die nicht-terminierende Berechnungen verursachen und so kein Resultat liefern. Wir definieren $\perp$ durch die Standard-Interpretation einer funktionalen Programmiersprache.

**Definition 2.1** *Sei $M$ die Menge der Ausdrücke einer funktionalen Sprache $L$, und sei $N$ die Menge der Normalformen in $M$. Die Standard-Interpretation ordnet dann jedem Ausdruck in $M$ seine Normalform zu. Wir fügen dem Universum $\perp$ hinzu und interpretieren alle Ausdrücke in $M$ ohne Normalform als $\perp$.*

Es ist nicht sinnvoll zu versuchen, Ausdrücke bis Normalform zu reduzieren, denn das würde verlangen, all die Teilausdrücke, die $\perp$ entsprechen, zu identifizieren. Man definiert also ein schwächeres Kriterium, die schwache Kopf-Normalform (engl. *weak head normal form*, daher kurz WHNF).

**Definition 2.2 (schwache Kopf-Normalform)** *Ein Ausdruck ist in* schwacher Kopf-Normalform*, wenn er von der Form*

$$S\ e_1\ \ldots\ e_n$$

*ist, wobei*

1. *$S$ eine Variable oder ein Datenobjekt ist oder*
2. *$S$ ein Funktionssymbol ist, so daß für kein $m \leq n$ $S\ e_1\ \ldots\ e_m$ ein Redex ist.*

Die wohl wichtigste Art von Werten für funktionale Programmierung bilden die Funktionen. Diese haben den gleichen Status wie andere Werte, können also auch als Argumente an andere Funktionen übergeben werden und das Ergebnis von Funktionen sein. Solche Funktionen nennt man dann Funktionen höherer Ordnung und diese ermöglichen einen sehr ausdrucksstarken Programmierstil.

Die Funktionsanwendung ist schließlich derart wichtig und häufig, daß sie durch einfaches Nebeneinanderschreiben dargestellt wird. Es bezeichnet also `f x` die Anwendung von `f` auf `x`. Um Funktionen mehrerer Argumente darstellen zu können verwendet man *Currying* (nach Haskell B. Curry). Dabei wird beispielsweise die Funktion `+` nicht als Funktion von zwei Zahlen aufgefaßt, die eine Zahl ergibt, sondern in `(+) x y` wird `((+) x)` als Funktion aufgefaßt, die eine Zahl auf eine andere Zahl abbildet. `(+)` wird also aufgefaßt als Funktion einer Zahl, welche ihrerseits eine Funktion ergibt, die eine Zahl auf eine Zahl abbildet. Dies erlaubt mit allen Funktionen umzugehen, als hätten sie höchstens ein Argument.

### 2.1.1 Pattern Matching

Viele moderne Programmiersprachen (funktionale oder andere) haben reichhaltige Typsysteme und die Typen werden gewöhnlich statisch geprüft, zum Teil sogar hergeleitet.

Die funktionalen Sprachen sehen häufig die Definition neuer Datentypen durch den Benutzer vor. Diese werden von den Autoren unterschiedlich bezeichnet: z. B. *structured types*, *algebraic types* oder *free data types*. Hier sollen sie strukturierte Typen heißen.

Die Definition eines *strukturierten Typs* gibt seine Konstruktoren an und die Typen ihrer Argumente, z. B. in Haskell:

$$\begin{array}{rcl} data\ T & = & C_1\ T_{1,1}\ \ldots\ T_{1,r_1} \\ & \mid & C_2\ T_{2,1}\ \ldots\ T_{2,r_2} \\ & & \vdots \\ & \mid & C_n\ T_{n,1}\ \ldots\ T_{n,r_n} \end{array}$$

wobei die $T_{i,j}$ Typen sind und die $C_i$ *Konstruktoren* der Stelligkeit $r_i$. $T$ kann aufgefaßt werden als markierte Vereinigung von Typen $T = T_1 + T_2 + \ldots + T_n$

mit $T_i = T_{i,1} \times T_{i,2} \times \ldots \times T_{i,r_i}$. Strukturierte Typen nennt man daher auch *Summentypen*, wenn $n > 1$, ansonsten *Produkttypen*.

Um nun mit Ausdrücken strukturierter Typen umgehen zu können, d. h. die Konstruktoren unterscheiden zu können und auf deren Argumente zugreifen zu können, verwendet man Patterns und Pattern Matching.

**Definition 2.3 (Pattern)** *Ein* Pattern $p$ *ist entweder*

- *eine* Pattern-Variable $v$
- *oder eine* Konstante $k$, *wie z. B. eine Zahl, ein Zeichen, eine bool'sche Konstante usw.*
- *oder ein* Konstruktor-Pattern *der Form* $(C\ p_1\ \ldots\ p_r)$, *wobei* $C$ *ein Konstruktor der Stelligkeit* $r$ *ist und die* $p_i$ *Patterns sind.*

*Ein Pattern, in dem keine Variable mehrfach auftritt, heißt* linear *und Patterns, bei denen* $C$ *ein Konstruktor eines Summentyps ist, heißen* Summen-Patterns. *Die anderen Konstruktor-Patterns heißen* Produkt-Patterns. *Ein* simples Pattern *ist eines, in dem nur Konstruktor-Patterns auftreten, deren* $p_i$ *Variable sind.*

Patterns müssen mit Werten verglichen werden. Der Versuch, ein Pattern mit einem Wert zu matchen, kann eines von drei Ergebnissen haben: er kann scheitern, er kann erfolgreich sein, wobei dann für jede Variable im Pattern eine Bindung entsteht, und er kann divergieren, d. h. $\perp$ ergeben.

**Definition 2.4 (Pattern Matching)** Pattern Matching *ist die Methode, ein Pattern mit einem Wert zu vergleichen, wobei das Pattern von links nach rechts und von außen nach innen untersucht wird. Weiterhin werden folgende Regeln angewandt:*

- *Ein Match von* $\perp$ *mit einem beliebigen Pattern divergiert, es sei denn, das Pattern besteht nur aus einer Pattern-Variable.*
- *Ein Match eines Wertes* $v$, $v \neq \perp$, *mit einem Konstruktor-Pattern scheitert, wenn die äußeren Konstruktoren verschieden sind. Sind die äußeren Konstruktoren gleich, ist das Ergebnis das Ergebnis des Matches von allen Sub-Patterns von links nach rechts: wenn diese alle erfolgreich sind, ist der ganze Match erfolgreich und der erste der scheitert oder divergiert ergibt ein Scheitern bzw. Divergieren des ganzen Matches.*

Für simple Konstruktor-Patterns bedeutet dies, daß sie einen Wert matchen, wenn die äußeren Konstruktoren übereinstimmen.

In nicht-strikt auswertenden funktionalen Sprachen kann Auswertung abweichend von Normalordnung nur durch Pattern Matching verursacht werden.

## 2.2 Definitionen und Notationen

**Definition 2.5 (Superkombinator, -Redex, -Reduktion)** *Ein* Superkombinator *$f$ der Stelligkeit $n$ ist ein Ausdruck*

$$f\ x_1\ \ldots\ x_n = e,$$

*wobei*

1. *$e$ nur freie Variablen aus $\{x_1, \ldots, x_n\}$ hat,*
2. *jede Funktionsanwendung in $e$ einen Superkombinator verwendet,*
3. *$n \geq 0$ ist, d. h. ein Superkombinator muß keine Argumente haben, nullstellige Superkombinatoren sind* konstante, applikative Formen*, kurz* CAF.

*Ein* Superkombinator-Redex*, kurz* Redex*, ist die Anwendung eines Superkombinators der Stelligkeit $n$ auf $n$ Argumente.* Superkombinator-Reduktion *ersetzt schließlich einen Superkombinator-Redex $s$ durch die rechte Seite des Superkombinators, in der die Argumente für jedes freie Auftreten der formalen Parameter substituiert wurden, $t$. Dafür schreiben wir $s \to t$.*

Eine wichtige Frage, die sich hier stellt, ist, ob Superkombinator-Reduktion die gleichen Antworten liefert, wie z. B. Graphreduktion. Lester hat in [Les88] gezeigt, daß dies so ist.

Die Auswertung von Ausdrücken durch den Computer können wir durch Reduktionen darstellen. Wir benötigen eine Datenstruktur, um die Ausdrücke während dieser Reduktionen zu speichern und eine Methode diese Struktur zu verändern, um die durchgeführten Reduktionen darzustellen. Bäume könnten für diesen Zweck verwendet werden, aber Graphen haben den Vorteil, daß Teilausdrücke, durch Teilgraphen dargestellt, an mehreren Stellen verwendet werden können und nur einmal ausgewertet werden müssen. Eine Reduktion erfolgt durch Überschreiben des Graphen durch seine reduzierte Form. Das Ausführen von Programmen auf diese Weise nennt man *Graphreduktion*.

Durch Graphen werden auszuwertende Terme repräsentiert. *Kontexte* sind besondere Terme, die *Lücken* oder *freie Stellen* enthalten. Einen Kontext mit $n$ Lücken schreibt man $C\underbrace{(\ ,\ldots,\ )}_{n}$. Für Terme $t_1, \ldots, t_n$ bezeichnet $C(t_1, \ldots, t_n)$ das Resultat nach Einsetzen der $t_1, \ldots, t_n$ in die Lücken von links nach rechts.

Betrachtet man die Superkombinatoren:

```
k a b = a
  h x = h x
```

kann der Ausdruck:

```
k 1 (h 1)
```

auf verschiedene Weisen reduziert werden.

Die eine ist:

$$\texttt{k 1 (h 1)} \rightarrow \texttt{1},$$

bei der anderen wird zunächst versucht `(h x)` zu reduzieren. Wir erhalten:

$$\begin{array}{c} \texttt{k 1 (h 1)} \rightarrow \texttt{k 1 (h 1)} \\ \vdots \end{array}$$

Die Bedeutung eines Programms ändert sich also, wenn wir die Reihenfolge verändern, in der Teilausdrücke ausgewertet werden. Es stellt sich heraus, daß man die erwartete Antwort erhält, wenn immer der *linkeste, äußerste* Redex reduziert wird. In diesem Fall spricht man von Reduktion in *Normalordnung*. Eine Abbildung $F$, die zu jedem Ausdruck $e$ einen Ausdruck $F(e)$ liefert, der durch Reduktion von $e$ erreicht werden kann, heißt *Reduktionsstrategie*. *Nicht-strikte* Auswertung bedeutet, daß

- in normaler Ordnung reduziert wird,
- Ausdrücke bis WHNF reduziert werden und
- jeder Ausdruck, der ein formaler Parameter einer Funktion ist, höchstens einmal ausgewertet wird.

Eine Reduktionsstrategie ist *sicher* für einen Ausdruck, wenn sie nur für den Fall, daß der Ausdruck keine WHNF hat, eine unendliche Berechnung startet. Wir sagen, daß es sicher ist, einen Ausdruck auszuwerten, wenn die resultierende Reduktionsstrategie sicher ist. Als *funktionale Strategie* bezeichnet man die Reduktionsstrategie, die sich ergibt, wenn man beim Pattern Matching immer von links nach rechts und von außen nach innen auswertend vorgeht. In Sprachen, die mehrere Ersetzungsregeln für eine Funktion erlauben, verlangt die funktionale Strategie die Regeln in der Reihenfolge, in der sie geschrieben sind zu betrachten.

### 2.2.1 Haskell und Gofer

Haskell ist eine universelle, rein funktionale Sprache, die viele kürzliche Innovationen aus dem Bereich der Programmiersprachen enthält. Dazu gehören:

- Funktionen höherer Ordnung,
- nicht-strikte Semantik,
- statische polymorphe Typisierung,
- benutzerdefinierte strukturierte Datentypen,
- Pattern Matching,

- die Mengenschreibweise,
- ein Modulsystem,
- und eine reichhaltige Sammlung primitiver Datentypen.

Die primitiven Datentypen umfassen Listen, Arrays, ganze Zahlen fester und beliebiger Genauigkeit und Fließkommazahlen. Der Haskell Report [HPW+92] sagt: "Haskell is both the culmination and solidification of many years of research on functional languages [..]"

Darüberhinaus beinhaltet Haskell ein Verfahren, um explizit Operatoren überladen zu können, das in das Typsystem integriert ist, Möglichkeiten Daten zu abstrahieren und Information zu verbergen, ein I/O-System, das referentiell transparent ist und eine „Arrayschreibweise".

Die meisten der Eigenschaften von Haskell finden sich auch in Gofer, dort fehlt jedoch das Modulkonzept und die Sammlung primitiver Datentypen ist etwas kleiner.

#### Pattern Matching

Haskell erweitert das Pattern Matching, wie es in 2.1.1 vorgestellt wurde, um die explizite Benennung von an Teilpatterns gebundenen Werten (as-patterns), Patterns, die nicht-strikt gematcht werden (irrefutable patterns), Patterns, die einen numerischen Wert matchen, wenn dieser ein Minimum $m$ übersteigt und die Pattern-Variable an die Differenz zu $m$ binden (sucessor patterns) sowie negative Literale im Pattern.

As-patterns werden im Programm im Anhang verwendet, irrefutable patterns werden in den I/O Routinen (nicht ausgedruckt) verwendet. Für eine Diskussion dieser Erweiterungen sei auf [HPW+92] und [Jon93] verwiesen.

### 2.2.2 Concurrent Clean

Concurrent Clean ist eine experimentelle rein funktionale Sprache mit nichtstrikter Auswertung, die für die Auswertung auf verschiedenen Maschinenarchitekturen, von einfachen Computern (Apple Macintosh) bis zu hochgradig parallelen Architekturen (Parsytec Supercluster) geeignet ist.

Concurrent Clean wurde als eine Sprache konzipiert, in die sich beliebige funktionale Sprachen übersetzen lassen und die sich ihrerseits leicht auf beliebige Maschinenarchitekturen abbilden läßt.

Concurrent Clean unterstützt nur die grundlegenden Aspekte funktionaler Sprachen. Syntaktischer Zucker (wie Infix-Notation oder komplexe Mengenschreibweisen) wurde weitgehend weggelassen. Concurrent Clean enthält aber ein Typsystem, hat eine Modulstruktur, eine moderne I/O Behandlung sowie Annotationen für Striktheits-Informationen und Prozeßtopologien.

#### Pattern Matching

Auch in Concurrent Clean gibt es „as patterns“ und „sucessor patterns“.

### 2.2.3 Der Sprachkern

Die „Striktheits-Analyse“, wie sie hier implementiert ist, operiert auf Programmen in einer sehr eingeschränkten Sprache, dem *Sprachkern*, dessen Syntax in Abbildung 2.2.3 zu sehen ist.

Dieser Sprachkern ist nicht geeignet, um größere Programme darin zu schreiben, aber er enthält alle Bestandteile, die notwendig sind, um Programme aus einer reichhaltigeren Programmiersprache, wie z. B. Haskell darzustellen. Tatsächlich stellt er, als „Core Syntax“, im Glasgow Haskell Compiler während einiger Transformationen das Programm dar. Ebenso wird er in [PJL91] als „Core Language“ verwendet. Dort wird argumentiert, daß die Beschränkungen nicht die Ausdruckskraft der Sprache oder die Effizienz der in sie übersetzten Programme beschränken. Sowohl als „Core Syntax“ als auch als „Core Language“ fehlen die syntaktischen Konstrukte `Top`, `Bot` und Vereinigungen, die zum experimentieren mit den Eigenschaften der abstrakten Reduktion in die Sprache eingeführt wurden. Ein Programm dieser Sprache, ein *Kernprogramm*, besteht aus einer Menge von Superkombinator-Definitionen, mit einer ausgezeichneten: `main`. Um ein Kernprogramm auszuführen, wertet man `main` aus.

#### Pattern-Matching

Wie bereits in 2.1.1 erwähnt, ist eine wesentliche Methode im funktionalen Programmieren die Definition von Datentypen und die Verwendung von Pattern-Matching.

Pattern-Matching kann z. B. in Haskell an verschiedenen Stellen auftreten: in Lambda-Ausdrücken, in Funktionsdefinitionen, in `where`- und `let`-Ausdrücken, in der Mengenschreibweise und in `case`-Ausdrücken. All diese Verwendungen von Pattern-Matching lassen sich in `case`-Ausdrücke übersetzen (siehe z. B. [PJ87] und [HPW+92]) und diese stehen auch im Sprachkern zur Verfügung.

Strukturierte Typen werden im Sprachkern nicht mehr unterschieden, es wird davon ausgegangen, daß das ursprüngliche Programm typgerecht ist. Die Konstruktoren innerhalb eines Typs werden durch eindeutige ganzzahlige Werte repräsentiert. Auf diese Weise ist es möglich, Objekte eines Typs, die mit verschiedenen Konstruktoren erzeugt wurden, zu unterscheiden.

Ein Beispiel hierfür sei der Typ `List`

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

Die Konstruktoren `Nil` und `Cons` würden im Sprachkern zu `Cons{1,0}` bzw. `Cons{2,2}`, wobei die erste Zahl für die Konstruktornummer steht und die zwei-

| | | | | |
|---|---|---|---|---|
| Programm | *prog* | $\rightarrow$ | $sk_1\ \texttt{;}\ \ldots\ \texttt{;}\ sk_n\ [\texttt{;}]$ | $n \geq 1$ |
| Super-kombinator | *sk* | $\rightarrow$ | $var\ var_1 \ldots var_n\ \texttt{=}\ expr$ | $n \geq 0$ |
| Ausdruck | *expr* | $\rightarrow$ | $expr\ aexpr$ | Anwendung |
| | | $\mid$ | $expr_1\ binop\ expr_2$ | Infix binäre Anwendung |
| | | $\mid$ | $\texttt{let}\ defs\ \texttt{in}\ expr$ | Lokale Definitionen |
| | | $\mid$ | $\texttt{letrec}\ defs\ \texttt{in}\ expr$ | Lokale rekursive Definitionen |
| | | $\mid$ | $\texttt{case}\ expr\ \texttt{of}\ alts$ | Case Ausdruck |
| | | $\mid$ | $\texttt{\textbackslash}\ var_1 \ldots var_n\ \texttt{.}\ expr$ | Lambda Abstraktion ($n \geq 1$) |
| | | $\mid$ | $aexpr$ | Atomarer Ausdruck |
| | *aexpr* | $\rightarrow$ | $var$ | Variable |
| | | $\mid$ | $num$ | Zahl |
| | | $\mid$ | $\texttt{Pack\{}num\texttt{,}num\texttt{\}}$ | Konstruktor |
| | | $\mid$ | $\texttt{(}\ expr\ \texttt{)}$ | Geklammerter Ausdruck |
| | | $\mid$ | $\texttt{Top}$ | $\top$ |
| | | $\mid$ | $\texttt{Bot}$ | $\bot$ |
| | | $\mid$ | $\texttt{<}expr_1, \ldots, expr_n\texttt{>}$ | Vereinigung |
| Definitionen | *defs* | $\rightarrow$ | $def_1\ \texttt{;}\ \ldots\ \texttt{;}\ def_n$ | $n \geq 1$ |
| | *def* | $\rightarrow$ | $var\ \texttt{=}\ expr$ | |
| Alternativen | *alts* | $\rightarrow$ | $alt_1\ \texttt{;}\ \ldots\ \texttt{;}\ alt_n$ | $n \geq 1$ |
| | *alt* | $\rightarrow$ | $\texttt{<}num\texttt{>}\ var_1 \ldots var_n\ \texttt{->}\ expr$ | $n \geq 0$ |
| Binärer Operator | *binop* | $\rightarrow$ | $arop \mid relop \mid boolop$ | |
| | *arop* | $\rightarrow$ | $\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$ | Arithmetik |
| | *relop* | $\rightarrow$ | $\texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{\~{}=} \mid \texttt{>=} \mid \texttt{>}$ | Vergleich |
| | *boolop* | $\rightarrow$ | $\texttt{\&} \mid \texttt{\textbar}$ | Bool'sch |
| Variable | *var* | $\rightarrow$ | $alpha\ varch_1 \ldots varch_n$ | $n \geq 0$ |
| Zahl | *num* | $\rightarrow$ | $ziffer_1 \ldots ziffer_n$ | $n \geq 1$ |
| | *alpha* | $\rightarrow$ | *ein alphabetisches Zeichen* | |
| | | $\mid$ | $\texttt{\_}$ | Unterstrich |
| | *varch* | $\rightarrow$ | $alpha \mid ziffer$ | |
| | *digit* | $\rightarrow$ | *ein numerisches Zeichen* | |

Abbildung 2.1: Syntax des Sprachkerns

te für seine Stelligkeit. In Patterns genügt es, nur die Konstruktornummer in <>-Klammern zu verwenden.

## 2.3 G-Machine

Einen Meilenstein in der Entwicklung von Maschinenmodellen für die Auswertung funktionaler Programme stellt sicherlich die G-Maschine dar. Die "große Idee" der G-Maschine ist, Ausdrücke in einem funktionalen Programm in G-Code zu übersetzen, der, wenn er ausgeführt wird, Instanzen der Ausdrücke erzeugt. Dies führt zu einer besonders schnellen Umsetzung der Graphreduktion.

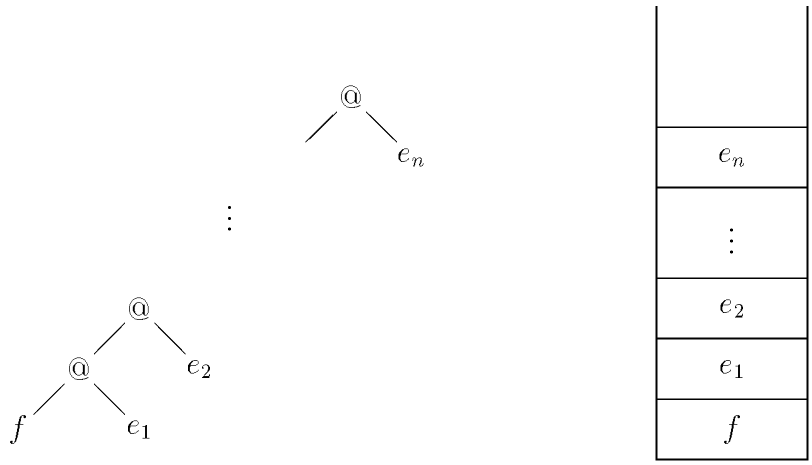
Die G-Maschine wurde an der Chalmers Tekniska Högskola in Göteborg, Schweden, von Johnsson [Joh84] und Augustsson [Aug84] entwickelt.

Sie versucht einige wesentliche Nachteile der Template Instantiierung zu beheben. Bei der Template Instantiierung wird die Template, also der Syntaxbaum, des Ausdrucks, rekursiv betrachtet und dabei ein Graph erzeugt, der eine Instanz des Ausdrucks ist. Bei der G-Maschine spart man also, wegen der bereits compilierten Superkombinatoren, für jedes Anlegen einer Instanz eines Ausdrucks das Durchwandern des Syntaxbaumes und die Entscheidung, welche Art Knoten jeweils verwendet werden muß. Die Template Instantiierung hat außerdem Einschränkungen, z. B. in bezug auf strukturierte Datentypen, die sie als Modell für allgemeine funktionale Programmiersprachen ungeeignet erscheinen lassen.

Die G-Maschine stellt mit ihren Varianten heute den Großteil der abstrakten Maschinen beim Compilieren funktionaler Programmiersprachen.

Hier sollen nur einige relevante Konzepte informell erklärt werden. In [PJ87], [Bur91], [Aug84], [Joh84] und [PJL91] können detailliertere Abschnitte über die G-Maschine gefunden werden. Eine formale Darstellung findet sich in [Les88], wo auch gezeigt wird, daß die G-Maschine eine korrekte Umsetzung der Graphreduktion ist.

Die G-Maschine benutzt im einfachsten Fall einen Stapel. Einige Weiterentwicklungen benutzen mehrere Stapel, jeweils für bestimmte Arten von Werten einen. Die Ausführung des Programmes beginnt mit einem Zeiger auf den Anfangsgraphen an der Stapelspitze. Der Anfangsgraph kann ein Knoten sein, der einen nullstelligen Superkombinator repräsentiert oder es kann sich um einen Anwendungsknoten handeln. Handelt es sich um einen Knoten für einen nullstelligen Superkombinator, dann kann sofort die erste seiner Anweisung angesprungen werden. Im Fall eines Anwendungsknotens zeigt ein Zeiger auf die anzuwendende Funktion und der andere auf deren Argument. Die anzuwendende Funktion kann ihrerseits wieder eine Anwendung sein. Den Pfad vom obersten Anwendungsknoten eines Redex zum anzuwendenden Superkombinator nennt man *Rückgrat* der Anwendung. Dieses muß bei Anwendungsknoten abgerollt werden, d. h. es muß ein Zeiger auf jeden Anwendungsknoten entlang des Rückgrats auf den Stapel gelegt werden und schließlich ein Zeiger auf den anzuwendenden Superkombinator. Abbildung 2.2 zeigt den Graphen einer Anwendung und den Stapel nach dem




(a) Graph der Anwendung von $f$ (b) Stapel nach dem abrollen von $f$

Abbildung 2.2: Das Rückgrat von $f$

Abrollen der Funktion.

Steht ein Zeiger auf einen Superkombinator $f$ an der Stapelspitze, folgt die Maschine diesem Zeiger und ermittelt dabei die Stelligkeit von $f$ sowie die Anfangsadresse der Anweisungen für $f$. Zuerst prüft sie, ob genug Argumente auf dem Stapel liegen, um $f$ auszuführen. Ist dies der Fall entfernt sie $f$ vom Stapel ordnet den Stapel etwas um und springt zur ersten Anweisung für $f$. Beim Umordnen werden die Zeiger auf die Anwendungsknoten durch Zeiger auf die Ausdrücke, die Argumente für $f$ sind, ersetzt. Die Maschine fährt dann mit den Anweisungen für $f$ fort.

Ein G-Maschinen Compiler wird für

```
fib n = if (n <= 1) 1 (fib (n-1) + fib (n-2))
```

folgenden G-Code erzeugen:

```
Pushint 2
Push 1
Pushglobal -
Mkap
Mkap
Pushglobal fib
Mkap
Pushint 1
Push 2
Pushglobal -
Mkap
Mkap
Pushglobal fib
Mkap
```

```
Pushglobal +
Mkap
Mkap
Pushint 1
Pushint 1
Push 3
Pushglobal <=
Mkap
Mkap
Pushglobal if
Mkap
Mkap
Mkap
Update 1
Pop 1
Unwind
```

Die Ausführung einer Graphersetzung von `fib` wird in Abbildung 2.3 Schritt für Schritt gezeigt. Dabei können wir beobachten:

1. daß beim Eintritt der Parameter `n` an der Stapelspitze ist und ein Zeiger auf die Wurzel des Redex direkt darunter
2. daß Einträge auf dem Stapel, die nicht an der Spitze stehen relativ zur Stapelspitze addressiert werden, wobei die Spitze die relative Adresse 0 hat. Eine Ausnahme ist hier die `Update` Anweisung, die die Spitze selbst nicht mitzählt, da diese nie ihr Ziel sein kann
3. daß sich einige Anweisungen wie Anweisungen einer Null-Adress-Maschine verhalten

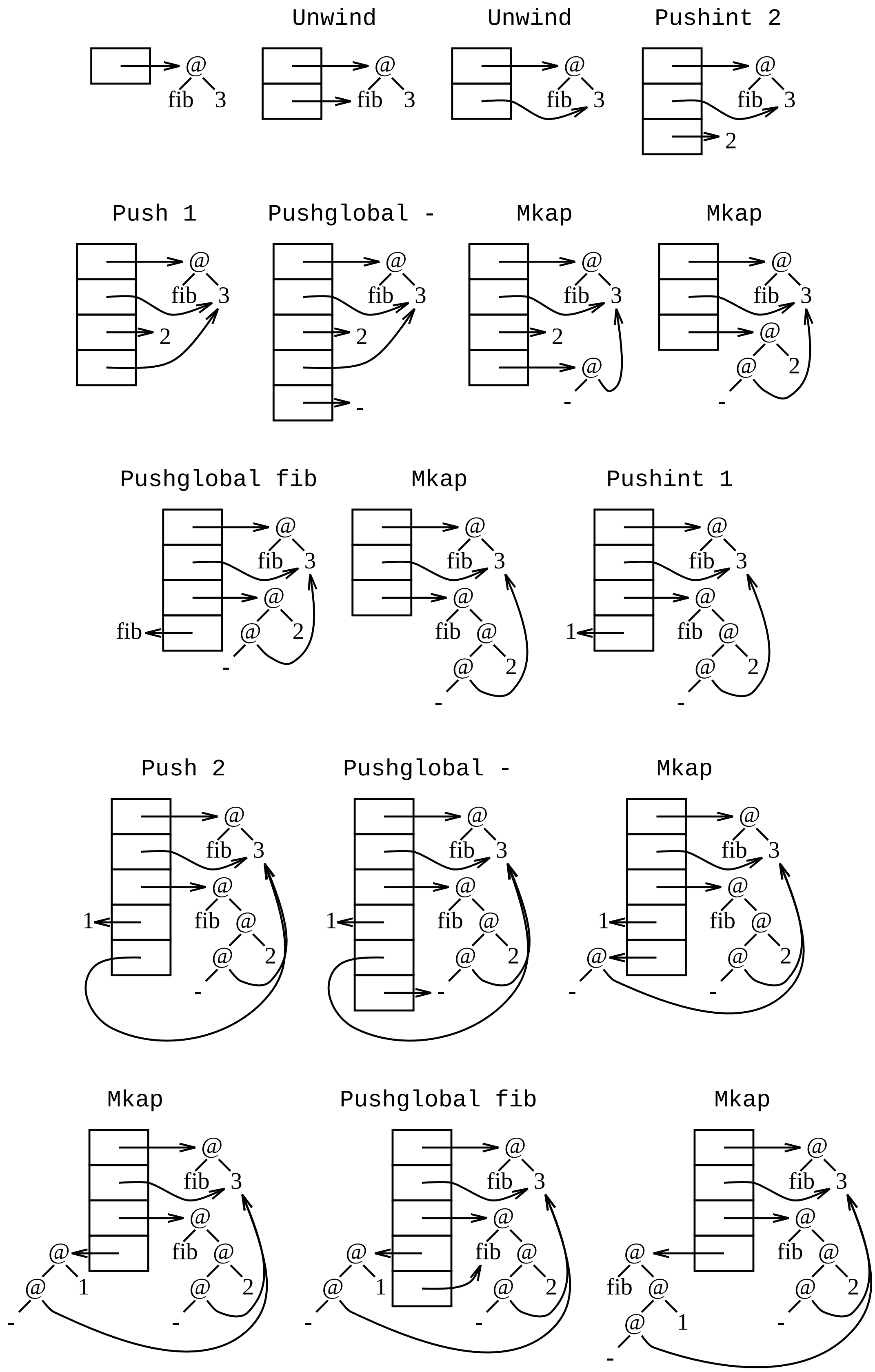
Die in den Abbildungen mehrfach auftretenden Symbole, wie beispielsweise `fib`, `1` oder `+` liegen tatsächlich nur einmal im Heap der G-Maschine vor, und alle Verweise beziehen sich auf dieses eine Objekt. Nur um die Übersichtlichkeit zu erhöhen, haben wir Symbole mehrfach in einer Abbildung verwendet. Abgesehen von den letzten drei Anweisungen konstruiert die Folge eine Instanz des Körpers von `fib`.

Die `Update 1` Anweisung überschreibt die Wurzel des Redex mit einem Verweis auf die Wurzel des Resultats.

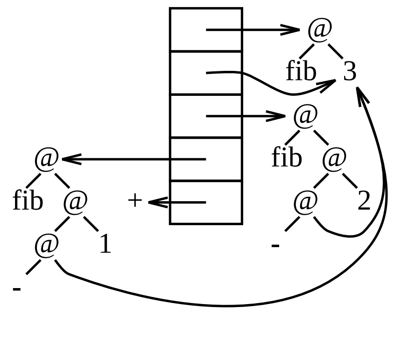
Die `Pop 1` Anweisung entfernt die Parameter der Funktion wieder vom Stapel, läßt also einen Stapel zurück, an dessen Spitze ein Zeiger steht, der auf die Wurzel des reduzierten Graphen verweist. Schließlich untersucht die `Unwind` Anweisung den Knoten, auf den der Zeiger von der Stapelspitze zeigt und fährt entweder fort oder beendet die Auswertung.

Ausgangspunkt für die Übersetzung in G-Code wird der Sprachkern sein. Das Übersetzen selbst ist für unsere Untersuchung weniger wichtig und soll daher hier

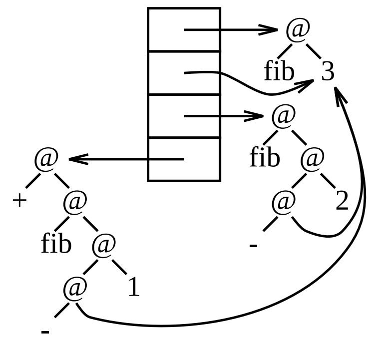
Abbildung 2.3: Graphersetzungsschritt bei `fib 3`



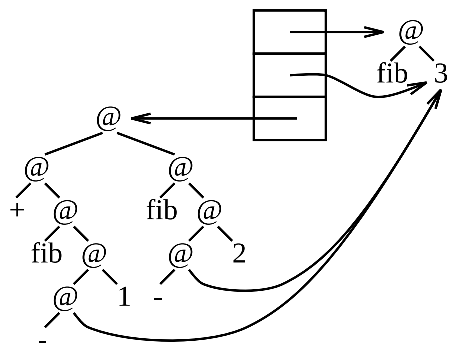
`Pushglobal +`



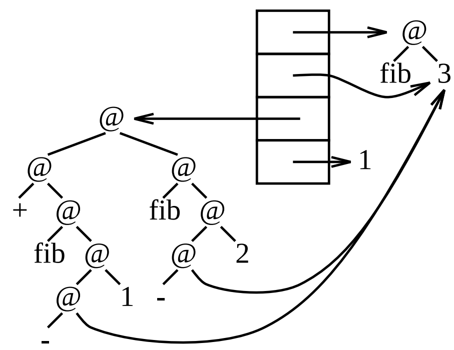
`Mkap`



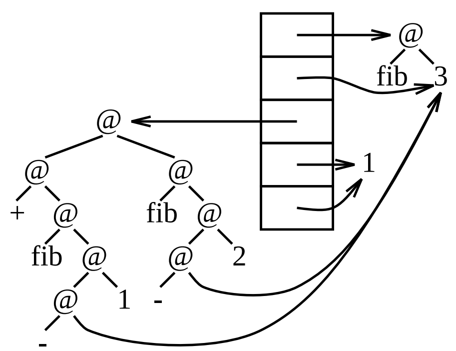
`Mkap`



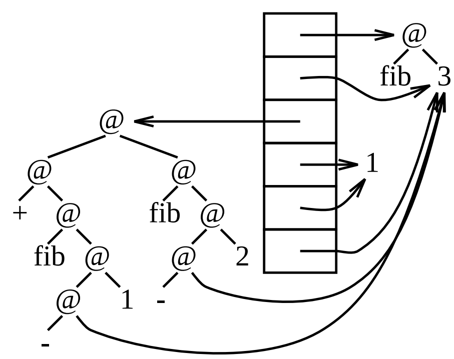
`Pushint 1`



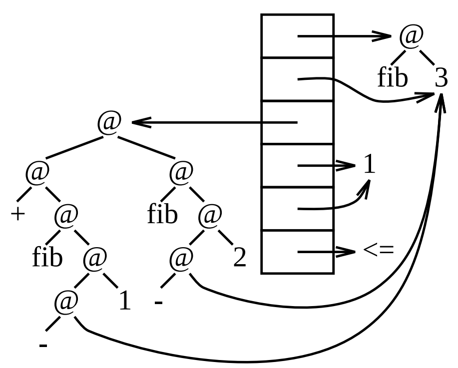
`Pushint 1`



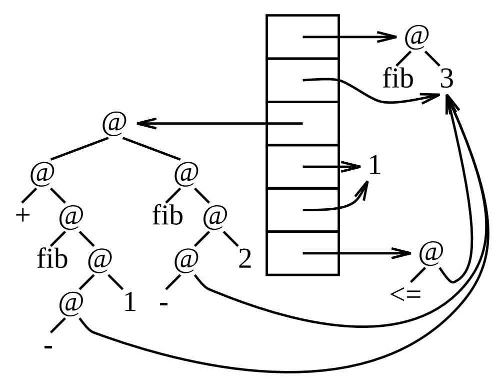
`Push 3`

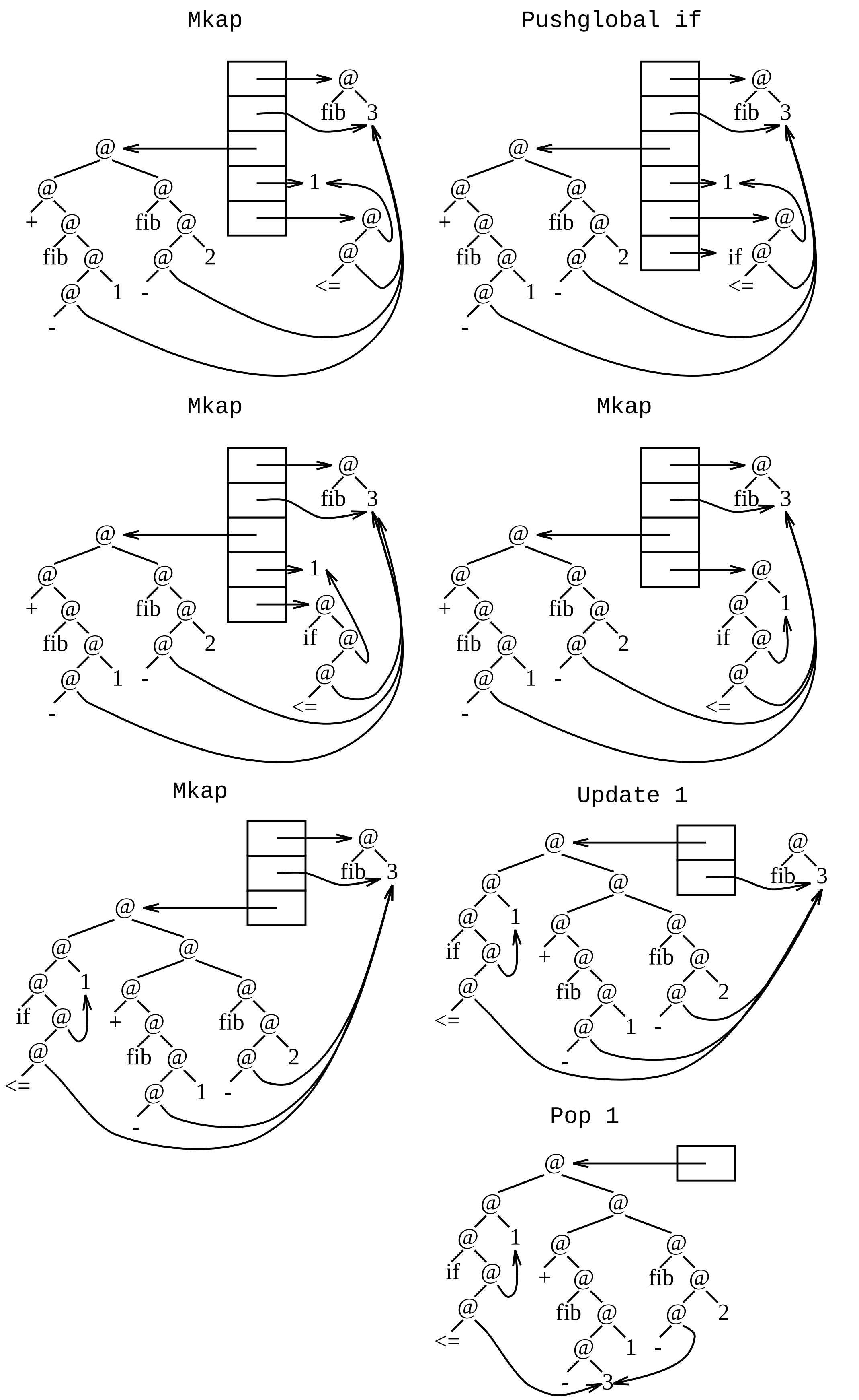


`Pushglobal <=`



`Mkap`




Mkap
Pushglobal if
Mkap
Mkap
Mkap
Update 1
Pop 1

nicht weiter beschrieben werden. [PJL91] und [PJ87] enthalten auf dieses Thema eingehende Kapitel.

Die G-Maschine, soweit sie hier kurz vorgestellt wurde, kann als ein Zustandsübergangssystem dargestellt werden, dessen Zustände 4-Tupel $\langle S, G, D, C\rangle$ sind, die aus den folgenden Komponenten bestehen.

S der Stapel

G der Graph

D der Dump

C der Code

Man beschreibt dann die Bedeutung der G-Code Anweisungen durch die Zustandsübergänge, die sie auslösen. Dies wird bei der Beschreibung der $G^{\#}$-Maschine getan.

Natürlich gibt es von der G-Maschine etliche Varianten und verwandte Modelle. U. a. die $\langle \nu, G\rangle$-Maschine, [AJ89a] und [AJ89b], wie sie die Sprache LML verwendet, ist ein paralleles Modell genau wie die Parallel G-Machine in [PJL91]. Die Spineless Tagless G-Machine [PJ93], eine Weiterentwicklung der Spineless G-Machine [Bur91]. Der Unterschied zur „Ur"-G-Maschine ist entweder effizienterer Code oder die Eignung für Parallelrechner oder beides. Es wäre sicherlich interessant zu untersuchen, inwieweit die abstrakte Reduktion und die Analyse des Reduktionspfades, wie sie im nächsten Kapitel besprochen werden, auf diese Modelle zu übertragen sind.

## 2.4 Striktheits-Analyse

In Abschnitt 2.2 haben wir nicht-strikte Auswertung und Normalordnung besprochen. Es stellt sich heraus, daß ein Abweichen von Reduktion in Normalordnung oft sinnvoll ist, weil durch frühzeitiges Auswerten von Teilausdrücken häufig die Laufzeit verkürzt werden kann. Ein einfaches Beispiel hierfür ist:

```
even x = x `rem` 2 == 0.
```

Da `rem` strict in beiden Argumenten ist, wird `x` schließlich ausgewertet. Jede Anweisung, die ausgeführt wird, um den Graphen für `x` im Heap anzulegen, kann gespart werden, wenn der Graph sofort ausgewertet wird und nur noch das Resultat im Heap angelegt wird. In 2.4.3 werden wir auf die zu erwartenden Einsparungen weiter eingehen. Was wir jedoch durch frühes Auswerten nicht möchten, ist

1. die Laufzeit erhöhen, indem Berechnungen durchgeführt werden, die schließlich nicht benötigt werden und

2. die Bedeutung des Programms verändern, indem wir einen nicht-terminierenden Ausdruck auswerten, der ebenfalls nicht benötigt wird.

In beiden Fällen würden wir Auswertungen durchführen, deren Ergebnis nicht benötigt wird.

Ein Kriterium dafür, ob eine Funktion ihr Argument benötigt, erhalten wir aus der Definition der Striktheit.

**Definition 2.6 (Striktheit)** *Eine Funktion $f$ heißt* strikt *in ihrem Argument, genau dann wenn sie auf ein nicht-terminierendes Argument angewandt selbst nicht terminiert, also:*

$$f \perp = \perp.$$

Für Funktionen, die für bestimmte Belegungen ihres Argumentes terminieren, folgt aus der Striktheit, daß die Funktion ihr Argument[1] benötigt. Funktionen, die für keine Belegung ihres Argumentes terminieren sind nach obiger Definition strikt in ihrem Argument. Selbst für diese kann das Argument frühzeitig ausgewertet werden, ohne seine Bedeutung zu verändern.

### 2.4.1 Unentscheidbarkeit

**Theorem 2.7** *Die Striktheits-Analyse ist unentscheidbar.*

Beweis: Angenommen es gäbe einen Algorithmus, der für jedes $f$, $f \perp = \perp$ entscheidet. Daraus erhält man einen Algorithmus für das Halteproblem, indem man eine Instanz $\langle M, w \rangle$ des Halteproblems durch eine Funktion $f_i$ darstellt. $f_i$ verwendet ein Argument und berechnet unabhängig von diesem das Ergebnis der Turing-Maschine $M$ auf Eingabe $w$. Auf $f_i$ wendet man den Algorithmus für die Striktheits-Analyse an. Man prüft also $f_i \perp = \perp$. Es folgt $f_i \perp = \perp \iff f_i$ hält nicht. Da das Halteproblem unentscheidbar ist und auf die Striktheits-Analyse reduzierbar ist, ist auch die Striktheits-Analyse unentscheidbar. □

Das soll uns nicht daran hindern, Striktheit zu analysieren, nur müssen wir eben approximieren.

### 2.4.2 Terminierungsanalyse

Aus der Reduktion des Halteproblems auf die Striktheits-Analyse folgt, daß eine Approximation der Striktheits-Analyse auch eine Approximation des Halteproblems liefert. Wir können also mit der hier verwendeten Methode untersuchen, ob

[1] Bei Verwendung von Currying genügt es von einem Argument zu sprechen. Hat die Funktion $f'$ $n$ Argumente und wir interessieren uns für Striktheit in $i$-ten, $1 \leq i \leq n$, können wir für die Funktion $f = (\lambda x_1 \ldots \lambda x_{i-1}.\lambda x_{i+1} \ldots \lambda x_n.f') \underbrace{\top \ldots \top}_{n-1}$ die Anwendung $f \perp = \perp$ untersuchen.

verwendete Ausdrücke nicht-terminierende Auswertungen verursachen. Für eine Funktion `nonTerm`:

```
nonTerm x  =  if (x == 4)
                 (nonTerm x)
                 x;
```

und einen Ausdruck `nonTerm 4` kann die verwendete Methode `nonTerm 4 =` $\perp$ ableiten. Solche Information kann man z. B. dazu nutzen, dem Programmierer zur Übersetzungszeit die nicht-terminierenden Ausdrücke aufzuzeigen, damit er diese kritischen Stellen erneut prüft. Der Zeitaufwand, der mit der verwendeten Methode hierfür entsteht, ist etwa genauso groß wie der für Striktheits-Analyse. Nicht-terminierende Funktionen, besonders solche, die für kein Argument terminieren sind äußerst selten. Es erscheint daher nicht ratsam, für Argumente, für die zunächst keine Striktheit gefunden wurde, zu prüfen, ob Striktheit wegen Nicht-Terminierung vorliegt.

### 2.4.3 Nutzen der Striktheits-Information

In der Literatur zu funktionalen Programmiersprachen findet sich häufig eine Aussage wie: „Es wäre also vorteilhaft, statisch feststellen zu können, ob ein Ausdruck strikt ausgewertet werden muß oder nicht.“ Selten wird besprochen, *wie* diese Information zum Vorteil angewandt wird. Eine Ausnahme bildet hier [PJP]. Wir wollen hier in Anlehnung an diese Darlegung etwas mehr ins Detail gehen und betrachten, welche Möglichkeiten wir haben, Striktheits-Information zu nutzen.

#### Ganze Zahlen

Nicht-strikt auswertende funktionale Sprachen müssen zwischen eingepackten und ausgepackten Werten unterscheiden. Eingepackte Werte sind geeignet für die Darstellung im Heap und für die Unterscheidung von Redexen und Werten in WHNF. Die Ausgepackten sind dann die Werte, auf denen tatsächlich gerechnet wird. Z. B. hat `Int` die Darstellung:

```
data Int  =  MkInt Int#
```

wobei `Int#` der Typ der ganzen Zahlen ist, auf denen gerechnet wird.

Nehmen wir an, `f` ist strikt, mit dem Typ `f :: Int -> Int` und weiter, daß wir für den Aufruf `f (x+1)` Code erzeugen wollen. Wenn wir nicht wissen, ob `f` strikt in seinem Argument ist, müssen wir für `x+1` einen Teilgraphen anlegen und ihn `f` übergeben. Wenn wir aber wissen, daß `f` strikt ist, können wir `x+1` vor dem Aufruf auswerten. Wir können sogar das Argument an `f` ausgepackt übergeben, da wir jetzt wissen, daß es sich mit Sicherheit um einen ausgewerteten `Int` handelt. Dafür benötigen wir jedoch eine andere Funktion, einen *Arbeiter*, der die tatsächliche Ver*arbeitung* von `f` auf ausgepackten Werten darstellt. Diese hat den Typ `f´ :: Int# -> Int`, wobei `Int#` der Typ der ausgepackten ganzen Zahlen ist. In einer Umsetzung würde ein `Int` durch einen Zeiger auf eine Heap-Zelle dargestellt, die entweder den noch zu evaluierenden Ausdruck oder das evaluierte Resultat enthält. Ein `Int#` wird durch die ganze Zahl selbst dargestellt, ohne jeden Zeiger.

Was gewinnen wir?

Wir legen keinen Graphen für `x+1` an und sparen dadurch:

- Test auf Heap-Überlauf
- Schreiben des Graphen in den Heap
- Lesen des Graphen aus dem Heap in `f`
- Überschreiben der Wurzel von `x+1` mit dem Resultat
- Amortisiert: Garbage-Collection des Graphen

In [PJP] werden hierfür Kosten von ca. 25 Anweisungen pro Argument angegeben. Wir erzielen also eine deutliche Einsparung, wenn wir diese Anweisungen durch eine einzige ersetzen, die das Argument in ein geeignetes Register lädt. Noch deutlicher wird diese Einsparung bei rekursiven Funktionen, denn hier kann der Arbeiter sich selbst aufrufen, mit dem Effekt, daß für jeden rekursiven Aufruf diese Einsparung erzielt wird.

Erzielen wir so immer eine Einsparung?

Ist das Argument an `f` ein Argument der aufrufenden Funktion, die in diesem nicht strikt ist, dann erzielen wir durch frühzeitiges Auswerten von `x` keine Einsparung. Wir können aber redundantes Auspacken einsparen, das stattfindet, wenn wir versuchen, einen bereits ausgewerteten Ausdruck auszuwerten, wie z. B. in: `g y x = case y of <1> -> ...; <2> -> f x + f x`. Hier kann `x` vor Übergabe an `f` ausgewertet werden und an beide Aufrufe von `f` ausgepackt übergeben werden. So sparen wir das redundante Auspacken von `x`, das sonst in einem der Aufrufe stattgefunden hätte.

Es ergeben sich zusammenfassend drei verschiedene Bereiche, in denen wir Striktheits-Information nutzen können.

1. Anlegen und Überschreiben von Teilgraphen vermeiden.
2. Ausgepackte Daten direkt verarbeiten.
3. Redundante Auswertungen vermeiden.

Welche wesentliche Eigenschaft in der Darstellung der ganzen Zahlen wird hier ausgenutzt?

Wenn ein Ausdruck vom Typ `Int` in WHNF ist, dann *muß* sein äußerster Konstruktor `MkInt` sein, denn das ist der einzige Konstruktor im Typ `Int`. Deshalb können wir, wenn wir wissen, daß der Ausdruck in WHNF ist, den Konstruktor ignorieren und uns sofort seinem Argument zuwenden.

### Andere Typen mit einem Konstruktor

Die Beobachtungen, die wir bei den ganzen Zahlen gemacht haben, lassen sich auf andere Datentypen, die nur einen Konstruktor zur Verfügung stellen, verallgemeinern. Beispiele hierfür sind `Int`, `Char` etc, aber auch Tupel und andere.

### Multi-Konstruktor-Typen

Bei Typen mit mehreren Konstruktoren sehen die Dinge nicht mehr so einfach aus. Angenommen wir wüßten von `h`, daß sie strikt in ihrem Listen-Argument ist:

```
h :: [Int] -> Bool
```

Sicherlich könnten wir zwei Arbeiter für `h` anlegen, wobei der eine für die leere Liste und der andere für Listen mit mindestens einem Element verwendet wird. Dies ist jedoch kein vielversprechender Ansatz, weil die Zahl der verschiedenen Arbeiter, die wir benötigen, exponentiell in der Zahl der Argumente mit Multi-Konstruktor-Typen ist. Eine andere Möglichkeit wäre die Übergabe von Argumenten variabler Größe vorzusehen, zusammen mit der Information, welcher Konstruktor dargestellt werden soll. Gegenüber der ersten Alternative scheint diese vielversprechender, stellt aber höhere Anforderungen an den Codegenerator. Das statische Auspacken von strikten Komponenten eines Multi-Konstruktor-Typs ist also recht komplex. Eine Ausnahme bilden die Aufzählungstypen (mehrere nullstellige Konstruktoren). Diese haben eine offensichtliche Darstellung als ganze Zahl.

### Frühe Auswertung von Multi-Konstruktor-Typen

Selbst wenn man ein Argument eingepackt weitergibt, kann es sein, daß durch frühe Auswertung etwas gespart werden kann. Dazu überlegen wir uns, was für einen Aufruf (`h <arg>`) passiert, wobei `<arg>` nicht Variable und nicht Konstante oder Konstruktoranwendung ist.

Ohne Striktheits-Information müssen wir einen Graphen für `<arg>` anlegen und an `h` übergeben. Wissen wir, daß `h` strikt ist, können wir `<arg>` direkt auswerten, einen entsprechenden Konstruktor-Knoten anlegen und diesen an `h` übergeben. Dabei sparen wir den Aufwand, den Graphen im Heap anzulegen und später wieder zu lesen, um ihn auszuwerten.

### Verbessern von `let`-Ausdrücken

Im Ausdruck `let x =` $e_1$ `in` $e_2$, mit einem $e_2$, das strikt in `x` ist, ist es ebenfalls günstiger, $e_1$ sofort auszuwerten, anstatt dafür zunächst einen Graphen auf dem Heap anzulegen. [PJP] nennen dies „`Let`-to-`Case` Transformation", denn man kann diese Transformation in der Sprache selbst, durch Übergang von

$$\texttt{let x = } e_1 \texttt{ in } e_2$$

zu

$$\texttt{case } e_1 \texttt{ of x -> } e_2$$

darstellen.

### Kontexte

Alles bisher über das Ausnutzen der Striktheits-Information Gesagte bezieht sich auf reine Striktheits-Information.

Es ist möglich, daß wir über Striktheits-Information hinaus wissen, wie ein Ausdruck verwendet wird, also in welchem *Kontext* er steht. Dann unterscheiden wir drei Fälle:

1. Für nicht-rekursive Typen mit einem Konstruktor, wie z. B. Tupel, sollten wir in den Konstruktor sehen können, um die Striktheit seiner Komponenten zu untersuchen. So sollten wir z. B. bei `f (x,y) = x+1` die Benutzung von `x` ausnutzen und nicht nur Striktheit im Tupel.

2. Für rekursive Typen wurde bereits viel theoretische Arbeit geleistet, die die Art der Auswertung einer Funktion mit der Art der Auswertung ihrer Argumente in Beziehung setzt, z. B. [Bur91]. Die Methode, die hier verwendet wird, ist auch für solche Untersuchungen geeignet, mehr noch, solche Untersuchungen können mit dem gleichen Aufwand durchgeführt werden wie die Striktheits-Analyse selbst. Im Zusammenhang mit der abstrakter Reduktion und Pfadanalyse könnten sie durch geeignete abstrakte Werte und geeignete Funktionen für die Art der Auswertung dargestellt werden. Um die so gefundenen Informationen bei der Code-Erzeugung zu nutzen, müßte man dem Code-Generator mitteilen, wie die entsprechenden Auswertungsarten auf die Maschine abzubilden sind. Dies ist eines der Themen, denen sich der Autor nach Abschluß dieser Arbeit gerne widmen würde, es ist jedoch nicht Gegenstand der vorliegenden Arbeit.

3. Kontext-Information über verschachtelten nicht-rekursiven Typen mit mehreren Konstruktoren kann möglicherweise ausgenutzt werden, aber dies ist ebenfalls nicht Gegenstand dieser Arbeit.

# Kapitel 3

# Methode

Die hier verwendete Methode ist die abstrakte Reduktion mit Pfadanalyse. Abstrakte Reduktion wird der konkreten Reduktion nachgebildet, verwendet aber besondere (abstrakte) Werte, die für Mengen von konkreten Werten stehen und setzt diese in Beziehung zueinander. Hier zeigt sich eine gewisse Verwandtschaft zu den Methoden für Korrektheitsbeweise in imperativen Sprachen, wie wir sie in 2.1 erwähnt haben, wobei Vor- und Nachbedingung den abstrakten Werten und der Programmteil auf den diese sich beziehen der angewandten Reduktion entsprechen.

Die Pfadanalyse betrachtet Folgen von Reduktionen, um anhand der darin auftretenden Ausdrücke zyklische oder nicht-terminierende Strukturen zu erkennen. Erst durch die Pfadanalyse können wir Striktheit von rekursiven Funktionen untersuchen.

## 3.1 Notwendigkeit

Ein wesentlicher Bestandteil der Pfadanalyse ist, zu entscheiden, ob die Reduktion eines abstrakten Teilterms benötigt wird oder notwendig ist. Notwendigkeit konkreter Terme wird in Bezug auf die Normalform definiert.

**Definition 3.1 (Notwendigkeit)** *Ein Redex $\Delta$ in einem Term $t = C(\Delta)$ wird* benötigt*, ist* notwendig*, wenn in jeder Reduktion, die zu Normalform führt, $\Delta$ oder eines seiner Residuen reduziert wird. Ein* notwendiger Reduktionsschritt *ist einer, in dem ein notwendiger Redex reduziert wird.*

Der Begriff der Residuen stammt von den orthogonalen Termersetzungssystemen, siehe [Nö92]. Eine formale Definition der Konzepte würde den Rahmen dieser Arbeit sprengen, daher wollen wir hier eine intuitive Darstellung geben, die für unsere Zwecke ausreicht.

Wird in einem Ausdruck $t$ ein Teilausdruck $\Delta$ reduziert, wobei der Ausdruck $t'$ resultiert, so können wir die Nachfolger eines beliebigen Ausdrucks $s$ folgendermaßen charakterisieren:

- sind $s$ und $\Delta$ identisch, bleiben keine Nachfolger
- ist weder $s$ echter Teilausdruck von $\Delta$ noch umgekehrt, ist $s$ der einzige Nachfolger
- ist $\Delta$ echter Teilausdruck von $s$, gibt es genau einen Nachfolger, der durch Reduktion von $\Delta$ in $s$ entsteht
- ist schließlich $s$ echter Teilausdruck von $\Delta$, dann ist $s$ Funktions- oder Konstruktor-Argument und tritt möglicherweise in $t'$ auf. Jedes Auftreten von $s$ in dem aus $s$ reduzierten Teil von $t'$ ist Nachfolger von $s$.

Nachfolger, die Redexe sind, heißen *Residuen*.

Es ist bekannt, daß Reduktion gemäß Normalordnung einen Term in Normalform erzeugt, sofern es einen gibt. Weiterhin wird bei einer solchen Reduktion zu Normalform zwischenzeitlich ein Term in WHNF erreicht, da ja immer der linkeste äußerste Redex reduziert wird. Außerdem werden bei Normalordnung nur notwendige Reduktionen durchgeführt.

**Theorem 3.2** *Bei Normalordnung werden nur notwendige Reduktionen durchgeführt.*

Beweis: Nehmen wir an es würde ein Reduktionsschritt durchgeführt, der nicht notwendig ist. Es wird also ein Redex $\Delta$ reduziert, so daß nicht in jeder Reduktion zu Normalform $\Delta$ oder eines seiner Residuen reduziert wird. Betrachten wir eine solche Reduktion $A$, in der weder $\Delta$ noch eines seiner Residuen reduziert wird. Sei $t = C(\Delta)$ der ursprünglich zu reduzierende Term. Setzen wir für $\Delta$ $\perp$ ein erhalten wir $t' = C(\perp)$ und mit der Reduktion $A$ eine Normalform für $t'$. Damit gelangen wir zu einem Widerspruch, denn die Reduktion $A$ reduziert $\Delta$ in $t$, also auch $\perp$ in $t'$, kann also keine Normalform für $t'$ finden. □

## 3.2 Abstrakte Reduktion

Obwohl wir die Methode der abstrakten Reduktion mit Superkombinator-Reduktion verwenden werden, ist sie nicht auf diese beschränkt, sondern kann z. B. auch für Graph- oder Termreduktion verwendet werden.

Die Reduktion $\rightarrow$ aus Definition 2.5 erweitern wir um $\rightarrow_\perp$:

$$C(t) \rightarrow_\perp C(\perp) \iff t \text{ hat keine WHNF bzgl. } \rightarrow$$

Mit $\rightarrow^+$ bezeichnen wir die transitive Hülle von $\rightarrow \cup \rightarrow_\perp$. Im folgenden sei

$$t \rightarrow_n s \iff t \rightarrow^+ s \wedge \text{ mindestens eine der Reduktionen wird benötigt.}$$

**Definition 3.3** *Sei $S$ die Menge von Symbolen zu einer bestimmten Menge von Funktionen und Konstruktoren. Dann ist die Menge der* abstrakten Symbole $S^{\#}$ *zu $S$ folgendermaßen definiert:*

$$S^{\#} = \{f^{\#} | f \in S\} \cup \{\bot^{\#}, \top^{\#}, \texttt{Union}^{\#}\}.$$

*Die Menge der* abstrakten Graphen $\mathcal{AG}$ *ist die Menge der gerichteten, geordneten Graphen über $S^{\#}$.*

Dann ist für jedes Symbol aus $S$ ein entsprechendes Symbol in $S^{\#}$. Ist $\mathcal{G}$ die Menge der geordneten, gerichteten Graphen über $S$, dann heißt eine Funktion $\alpha : \mathcal{G} \to \mathcal{AG}$ *Abstraktion* und eine Funktion $\gamma : \mathcal{AG} \to P(\mathcal{G})$ heißt *Konkretisierung*. Die Konkretisierung wird zunächst für Baum-artige Graphen definiert (siehe [Nö93]).

$$\begin{array}{lcl}
\gamma_t : \mathcal{AG}_t \to P(\mathcal{G}) & & \\
\gamma_t(\bot^{\#}) & = & \{\bot\} \\
\gamma_t(\top^{\#}) & = & \mathcal{G} \\
\gamma_t(f\ x_1\ \ldots\ x_n) & = & \{f\ t_1\ \ldots\ t_n | (\forall 1 \leq i \leq n) : t_i \in \gamma_t(x_i) \\
 & & \qquad \wedge\ f \text{ ist Funktions- oder Konstruktorsymbol}\} \\
\gamma_t(\langle x_1, \ldots, x_n \rangle) & = & \bigcup_{1 \leq i \leq n} \gamma_t(x_i)
\end{array}$$

Dann kann man $\gamma(g) = \{h \in \mathcal{G} | U(h) \in \gamma_t(U(g))\}$ definieren, wobei $U(g)$ durch ausrollen von $g$ erzeugt wird. Wenn es der Zusammenhang erlaubt wird das # weggelassen. Die abstrakten Symbole $\top^{\#}$ und $\bot^{\#}$ stehen für $\top$ bzw. $\bot$ und sind nullstellig. $\texttt{Union}^{\#}$ steht für die Vereinigung abstrakter Graphen und liegt tatsächlich für jede Stelligkeit vor. Um $\texttt{Union}^{\#}$ abzukürzen schließen wir die Argumente in spitze Klammern $\langle\ \rangle$ ein. Das Zeichen @ verwenden wir, um Teilgraphen benennen zu können. Zyklische Strukturen, die wir mit `letrec` bilden, z. B. `letrec` $x =$ `Cons{2,2}` $1\ x$ `in` $x$, kürzen wir ab zu $x$@(`Cons{2,2}` $1\ x$). *Zyklische Vereinigungen* sind dann Ausdrücke der Form $x@\langle \ldots, x, \ldots \rangle$, auf die wir im Zusammenhang mit Pattern Matching weiter eingehen.

Die abstrakte Reduktion soll in erster Linie einige notwendige Reduktionen der entsprechenden konkreten Reduktion nachahmen. Diese Forderung führt uns zu folgender:

**Definition 3.4 (abstrakte Reduktion)** *Eine Relation $\to_\alpha$ auf $\mathcal{AG}$ heißt* abstrakte Reduktion, *genau dann wenn von jeder Konkretisierung des linken Graphen eine Konkretisierung des rechten Graphen mit einer benötigten Reduktion erreicht werden kann. Also:*

$$s \to_\alpha t \iff (\forall g \in \gamma(s))(\exists h \in \gamma(t)) : g \to_n h$$

Hierbei werden benötigte Reduktionen verwendet, um später sicher $\bot$ einführen zu können. Wenn aus dem Zusammenhang erkenntlich ist, daß abstrakte Reduktion gemeint ist, schreiben wir auch $\to$ statt $\to_\alpha$.

Wir haben in 2.2 Sicherheit aingeführt. Sicherheit läßt sich auch für abstrakte Reduktionen formulieren, dabei ist eine Reduktionsstrategie für abstrakte Reduktion sicher für einen (abstrakten) Ausdruck $a$, wenn sie $\bot^{\#}$ nur für den Fall reduziert, daß kein Ausdruck mit WHNF von $a$ repräsentiert wird.

### Beispiele

Man kann für einige Beispiele folgende Funktion betrachten:

```
f x n  =  case x of
              <1> -> n;
              <2> a b -> f b n;
```

Beispiele:

**Approximation:**

$$\texttt{f}\ \top\ \bot\ \rightarrow_\alpha\ \top$$

ist ein Beispiel für eine Approximation einer abstrakten Reduktion. Es ist sicher einen beliebigen abstrakten Graphen mit $\top$ zu approximieren, man verliert dabei jedoch Information.

**Ersetzung von `f`:** Sei `inf` definiert als `let inf = Cons{2,2} 1 inf in inf`, dann ergibt sich

$$\texttt{f inf}\ \top\ \rightarrow_\alpha\ \texttt{f inf}\ \top$$

aus den Definitionen von `inf` und `f`.

**Ersetzung von `f`:** Ebenso ergibt sich

$$\texttt{f}\ \top\ \bot\ \rightarrow_\alpha\ \langle\bot\texttt{,}\ \texttt{f}\ \top\ \bot\rangle$$

aus der Definition von `f`.

**Ersetung von `(+ 1 2)`:**

$$\begin{aligned}&\texttt{f (Cons\{2,2\} (+ 1 2) Cons\{1,0\}) (+ 1 2)}\\ &\quad\rightarrow_\alpha\ \texttt{f (Cons\{2,2\} 3 Cons\{1,0\}) 3}\end{aligned}$$

ist eine abstrakte Reduktion, weil `(+ 1 2)` an einer benötigten Position steht.

**Ersetzung von `(+ 1 2)`:**

$$\begin{aligned}&\texttt{f (Cons\{2,2\} (+ 1 2) Cons\{1,0\}) 3}\\ &\quad\not\rightarrow_\alpha\ \texttt{f (Cons\{2,2\} 3 Cons\{1,0\}) 3}\end{aligned}$$

hier steht `(+ 1 2)` nicht an einer benötigten Position, kann also auch nicht abstrakt reduziert werden.

**$\bot$-Einführung:**

$$\texttt{f}\ \top\ \bot\ \rightarrow_\alpha\ \bot$$

erfordert Pfadanalyse (siehe 3.4).

Striktheit in der abstrakten Welt wird dann so ausgedrückt:

$$f \perp = \perp$$

Wir erhalten also einen Begriff von schwacher Kopf-Normalform in der abstrakten Welt. Zu den abstrakten Werten, die nicht weiter reduziert werden sollen, gehören die, die konkrete Werte darstellen, die bereits in WHNF sind. Außerdem wollen wir, daß $\perp^\#$ in abstrakter WHNF ist. Es ergibt sich, daß ein abstrakter Wert in WHNF ist, wenn er

1. $\perp^\#$ ist,
2. $\top^\#$ ist,
3. ein Wurzelsymbol hat, welches ein Konstruktor ist oder
4. eine Vereinigung ist, die eine Komponente in WHNF enthält.

Der letzte Punkt kommt daher, daß eine Vereinigung, die eine Komponente in WHNF enthält, bereits konkrete Ausdrücke repräsentiert, die in WHNF sind, es ist daher nicht sicher in ihr weitere Komponenten zu reduzieren, denn diese Komponenten könnten nicht-terminierende Berechnungen verursachen.

### 3.2.1 $\leq$ auf $\mathcal{AG}$

Auf den abstrakten Werten können wir eine Ordnung definieren.

**Definition 3.5** ($\leq_\alpha$) *Für abstrakte Terme $s, t \in \mathcal{AG}$ gilt*

$$s \leq_\alpha t \iff \gamma(s) \subseteq \gamma(t).$$

Intuitiv sind die kleineren abstrakten Terme diejenigen, die mehr Information enthalten, denn sie stehen für eine kleinere Zahl von konkreten Termen.

Mit Hilfe dieser partiellen Ordnung läßt sich eine Äquivalenzrelation auf $\mathcal{AG}$ definieren:

**Definition 3.6** ($\equiv_\alpha$) *Für abstrakte Terme $s, t \in \mathcal{AG}$ gilt:*

$$s \equiv_\alpha t \iff s \leq_\alpha t \wedge t \leq_\alpha s$$

Die entstehenden Äquivalenzklassen haben in einigen Fällen Repräsentanten, die eine einfachere Struktur aufweisen als andere Repräsentanten. Dazu folgende Beispiele:

$$
\begin{aligned}
\langle x \rangle &\equiv_\alpha x && (3.1)\\
\langle x_1, \ldots, x_n \rangle &\equiv_\alpha \langle \pi(x_1, \ldots, x_n) \rangle && (3.2)\\
&\phantom{\equiv_\alpha}\ \text{für beliebige Permutationen } \pi\\
\langle x, x_1, \ldots, x_n \rangle &\equiv_\alpha \langle x_1, \ldots, x_n \rangle && (3.3)\\
&\phantom{\equiv_\alpha}\ \text{wenn } (\exists 1 \leq i \leq n) : x \leq_\alpha x_i\\
C(\langle x_1, \ldots, x_n \rangle) &\equiv_\alpha \langle C(x_1), \ldots, C(x_n) \rangle && (3.4)\\
x @ \langle x, x_1, \ldots, x_n \rangle &\equiv_\alpha \langle x_1, \ldots, x_n \rangle && (3.5)\\
\langle \langle x_1, \ldots, x_n \rangle, x'_1, \ldots, x'_m \rangle &\equiv_\alpha \langle x_1, \ldots, x_n, x'_1, \ldots, x'_m \rangle && (3.6)
\end{aligned}
$$

Diese Äquivalenzen erlauben uns mit Vereinigungen zu operieren und zu versuchen die Größe des zu betrachtenden Berechnungsbaums gering zu halten. (3.1) und (3.2) sprechen für sich, (3.3) erlaubt alle Komponenten außer den größten von maximalen Ketten wegzulassen, (3.4) beschreibt wie Vereinigungen in Termen verwandt werden und (3.5) erlaubt Zyklen zu entfernen. Das sieht man so:

$$
\begin{aligned}
\langle x_1, \ldots, x_n \rangle &\leq_\alpha x @ \langle x, x_1, \ldots, x_n \rangle \text{ trivial}\\
x @ \langle x, x_1, \ldots, x_n \rangle &\leq_\alpha \langle x_1, \ldots, x_n \rangle\\
\Longleftrightarrow \gamma(x @ \langle x, x_1, \ldots, x_n \rangle) &\subseteq \gamma(\langle x_1, \ldots, x_n \rangle)\\
\Longleftrightarrow \underbrace{y \cup \bigcup_{1 \leq i \leq n} \gamma(x_i)}_{y} &\subseteq \textstyle\bigcup_{1 \leq i \leq n} \gamma(x_i)
\end{aligned}
$$

3.6 ergibt sich folgendermaßen:

$\langle \langle x_1, \ldots, x_n \rangle, x'_1, \ldots, x'_m \rangle \leq_\alpha \langle x_1, \ldots, x_n, x'_1, \ldots, x'_m \rangle$ :

Wenn $x \in \gamma(\langle \langle x_1, \ldots, x_n \rangle, x'_1, \ldots, x'_m \rangle)$, dann ist $x \in \gamma(\langle x_1, \ldots, x_n \rangle)$ oder $(\exists 1 \leq i \leq m) : x \in \gamma(x'_i)$. Ist $x \in \gamma(x'_i)$ folgt $x \in \langle x_1, \ldots, x_n, x'_1, \ldots, x'_m \rangle$. Ist andererseits $x \in \gamma(\langle x_1, \ldots, x_n \rangle)$ dann $(\exists 1 \leq j \leq n) : x \in \gamma(x_j)$ und es gilt ebenfalls $x \in \langle x_1, \ldots, x_n, x'_1, \ldots, x'_m \rangle$. Die andere Richtung sieht man ganz genau so leicht.

Mit den Äquivalenzen können wir Vereinigungen in eine möglichst einfache äquivalente Form verwandeln. Dazu definieren wir, was wir unter „einfachen Vereinigungen“ zu verstehen haben. Zunächst definieren wir jedoch „\ für Vereinigungen“ und „Maxima von Vereinigungen“, die beide für die einfachen Vereinigungen benötigt weden.

**Definition 3.7** *Sei*

$$\langle x_1, \ldots, x_n \rangle \setminus x := \langle x_{k_1}, \ldots, x_{k_s} \rangle,$$

*so daß* $\{x_{k_1}, \ldots, x_{k_s}\} = \{x_1, \ldots, x_n\} \setminus x$ *ist.*

**Definition 3.8 (Maxima)** *Bezeichne*

$$M(\langle x_1, \ldots, x_n \rangle) := \begin{cases} \langle x_{m_1}, \ldots, x_{m_r} \rangle & r \geq 2 \\ x_{m_r} & r = 1 \end{cases}$$

*mit* $(\forall 1 \leq i \leq r) : x_{m_i} \leq_\alpha x_{m_j} \Rightarrow i = j$. $M(\langle x_1, \ldots, x_n \rangle)$ *sind die* Maxima *von* $\langle x_1, \ldots, x_n \rangle$.

**Definition 3.9 (einfache Vereinigung)** $M(x@\langle x_1, \ldots, x_n \rangle) \setminus x)$ *ist dann die* Vereinfachung *von* $\langle x_1, \ldots, x_n \rangle$. Einfache Vereinigungen *sind solche, die bereits vereinfacht sind.* Einfache (abstrakte) Werte *sind abstrakte Werte, in denen nur einfache Vereinigungen auftreten.*

Eine Eigenschaft einfacher Werte, die wir später noch benötigen werden, ist die folgende.

**Theorem 3.10** *Einfache Vereinigungen sind nicht zyklisch.*

Beweis: $x@\langle x_1, \ldots, x_n \rangle \setminus x$ ist keine zyklische Vereinigung.

Angenommen $M(\langle x_1, \ldots, x_n \rangle) = \langle x_{m_1}, \ldots, x_{m_r} \rangle$ wäre eine zyklische Vereinigung, dann gäbe es ein $i$, so daß $x_{m_i} = \langle x_{m_1}, \ldots, x_{m_r} \rangle$ ist. Es würde gelten $(\forall 1 \leq j \leq r) : x_{m_j} \not\leq_\alpha x_{m_i}$ im Widerspruch zur Definition von $M(\cdot)$.

### 3.2.2 Berechnung abstrakter Reduktionen

Die allgemeine Form der abstrakten Reduktion, wie wir sie bisher definiert haben ist viel zu allgemein, um implementiert zu werden. Sie erfordert zu entscheiden, ob zwei abstrakte Ausdrücke in einer Relation $\rightarrow_\alpha$ zueinander stehen, bei der für möglicherweise unendlich viele konkrete Graphen $g$ die Existenz eines konkreten Graphen $h$ überprüft werden muß, zu dem ein Pfad führt, der benötigte Reduktionen enthält, also $g \rightarrow_n h$. Das dabei notwendige Überprüfen der Notwendigkeit einer Reduktion ist offensichtlich unentscheidbar und wir können daher abstrakte Reduktion in dieser allgemeinen Form nicht verwenden.

In diesem Abschnitt werden wir besprechen, wie eine Lösung dieses allgemeinen Problems approximiert werden kann. Dies werden wir in zwei Schritten tun. Wir werden einen abstrakten Graphreduktionsmechanismus beschreiben, der in direkter Beziehung zu den konkreten Funktionsdefinitionen (und zur verwendeten Reduktionsstrategie) steht. Diese abstrakten Ersetzungen sind abstrakte Reduktionen. Später behandeln wir die Pfadanalyse, bei der durch Analyse von Folgen abstrakter Reduktionen neue abstrakte Reduktionen abgeleitet werden.

#### Abstrakte Graphersetzung

Betrachten wir den abstrakten Ausdruck

$$t = f^\# \ v_1 \ \ldots \ v_n$$

mit dem Funktionssymbol $f$. Dieser Term stellt die Menge $\gamma(t)$ von konkreten Graphen dar, die alle das Wurzelsymbol $f$ haben. Diese Terme werden also gemäß der Regel für $f$ ersetzt. Der abstrakte Ersetzungsschritt sollte alle konkreten Ersetzungsschritte simulieren. In [Nö93] treten hierbei zwei Schwierigkeiten auf:

1. verschiedene Ausdrücke in $\gamma(t)$ können gemäß verschiedener Alternativen von $f$ ersetzt werden und

2. für verschiedene Ausdrücke in $\gamma(t)$ kann die funktionale Strategie unterschiedliche Teilausdrücke zur Auswertung zwingen.

Im Sprachkern gibt es zu jedem Funktionssymbol höchstens eine Ersetzungsregel. Damit ist die erste Schwierigkeit überwunden. Der zweite Punkt führt in [Nö93] zu Schwierigkeiten, weil er die Reduktion und das Pattern Matching wesentlich verkompliziert. Da wir jedoch im Sprachkern nur simple Patterns zulassen und diese auch nur im `case` Konstrukt, stellt dieser Punkt hier keine echte Schwierigkeit dar. Die Ersetzungen können wie gewohnt vorgenommen werden. Wird jedoch ein Ausdruck durch ein `case` Konstrukt zur Auswertung gezwungen, muß beim Matching auf $\top$ und Vereinigungen besonders eingegangen werden. Im Fall von $\top$ muß jede Pattern Variable an $\top$ gebunden werden. Für Vereinigungen wird es etwas komplizierter.

Matchen mehrere Komponenten mit einer Alternative, dann müssen die Variablen auf verschiedene Arten gebunden werden. Man kann diesen Fall als Menge von Ersetzungen auffassen: für jeden möglichen Match der Vereinigung soll eine Ersetzung stattfinden. Betrachten wir folgendes Beispiel:

```
f x  =  case x of
               <2> a b -> Cons{3,2} a b
```

dann wird `f` $\top$ zu `Cons{3,2}` $\top$ $\top$ reduziert und

```
f <Cons{2,2} ⊤ ⊥, Cons{2,2} ⊥ ⊤> ⊢ <Cons{3,2} ⊤ ⊥, Cons{3,2} ⊥ ⊤>.
```

Obige Ersetzung kann man als Vereinigung zweier Ersetzungen auffassen, da

```
f <Cons{2,2} ⊤ ⊥, Cons{2,2} ⊥ ⊤>
  ≡α  <f (Cons{2,2} ⊤ ⊥), f (Cons{2,2} ⊥ ⊤)>
```

ist. Wenn ein Term mehrere Vereinigungen enthält, können die Ersetzungen recht kompliziert werden. Es erscheint daher ratsam Vereinigungen zu vereinfachen, sobald sie angelegt oder verändert werden. Schließlich bemerken wir, daß das Resultat der Ersetzung $\bot$ ist, wenn keine der Alternativen zutrifft (da `<>` $\equiv \bot$).

### Pattern Matching und die funktionale Strategie

In 2.1.1 haben wir dargelegt, daß Pattern-Matching das einzige Mittel ist Auswertung abweichend von Normalordnung zu verursachen. Es ist naheliegend, daß es für die Striktheits-Analyse von zentraler Bedeutung ist, aus Pattern-Matching Striktheits-Information zu gewinnen. Für die abstrakte Reduktion heißt das, wir müssen uns einen Matching Mechanismus konstruieren, der entscheidet welche Alternativen matchen (siehe auch [Nö93]). Intuitiv matcht ein Wert $w$ in $\mathcal{AG}$ ein Pattern $p$, wenn die Menge der konkreten Graphen, die $p$ repräsentiert in $\gamma(w)$, den Konkretisierungen von $w$, enthalten ist. Unser Entscheidungs-Algorithmus für abstraktes Matching ist sicher, denn nur wenn kein Wert aus $\gamma(w)$ mit dem konkreten Algorithmus matcht, matcht auch der abstrakte Algorithmus nicht.

$$\neg\text{Match}^{\#}(w, p^{\#}) \Rightarrow (\forall g \in \gamma(w)) : \neg\text{Match}(g, p)$$

soll also gelten. Dabei sind $p^{\#}$ und $p$ gleich, bis auf das Superskript # an den Symbolen.

Betrachten wir zunächst den Fall der Patterns im allgemeinen, also einen Entscheidungs-Algorithmus für abstraktes Matching, wie er für Clean, Gofer oder Haskell geeignet wäre und der unsere Forderung nach Sicherheit erfüllt. Ein einfacher Algorithmus ist Match$'$:

$$\begin{array}{lcl}
\text{Match}'(v, p) & = & \text{Wahr, wenn } p \text{ Variable ist} \\
\text{Match}'(\top, p) & = & \text{Wahr} \\
\text{Match}'(\bot, p) & = & \text{Falsch} \\
\text{Match}'(C\ w_1\ \ldots\ w_n, C'\ p_1\ \ldots\ p_m) & = & \text{Falsch, wenn } C \neq C' \\
\text{Match}'(C\ w_1\ \ldots\ w_n, C\ p_1\ \ldots\ p_n) & = & \bigwedge\{\text{Match}'(w_i, p_i) | 1 \leq i \leq n\} \\
\text{Match}'(\langle w_1\ \ldots\ w_n \rangle, p) & = & \bigvee\{\text{Match}'(w_i, p) | 1 \leq i \leq n\}
\end{array}$$

Match$'$ ist ungenau: Die funktionale Strategie des konkreten Matching wird nicht berücksichtigt. Betrachten wir dazu

```
f []  =  []
f x   =  [x]
```

Der Ausdruck `f []` würde mit Match$'$ folgendermaßen reduziert:

$$\texttt{f []} \rightarrow_\alpha \langle \texttt{[]}, \texttt{[[]]} \rangle,$$

weil beide Alternativen matchen. Und `f` $\bot$ würde reduziert zu

$$\texttt{f}\ \bot \rightarrow_\alpha \texttt{[}\bot\texttt{]}.$$

Beide Ersetzungen sind Sicher , aber auch zu pessimistisch. Im ersten Fall hätte uns die funktionale Strategie beim konkreten Matching garantiert, daß nur die erste Alternative matcht und im zweiten Fall hätte der konkrete Algorithmus nicht terminiert, da $\bot$ keine WHNF hat also wäre `f` $\bot = \bot$.

Es gilt also Match$'$ zu verbessern, so daß er dem konkreten Algorithmus besser entspricht. Dazu präzisiert man zunächst die Antwort, die vom Algorithmus gegeben werden kann. Statt nur zwei Werten erlauben wir vier Werte als Antwort. $\equiv$: ein Pattern matcht einen Wert vollständig, die anderen Alternativen sind dann uninteressant, weil die funktionale Strategie nur diese auswählen würde. $\subseteq$: ein Pattern matcht einen Wert teilweise, es gibt also Konkretisierungen des Wertes, die

das Pattern nicht matchen, diese matchen möglicherweise ein anderes Pattern. $\not\subseteq$: ein Pattern matcht einen Wert überhaupt nicht, es gibt also keine Konkretisierung des Wertes, die das Pattern matcht. $\perp$: der Wert der gematcht werden soll hat keine WHNF, der konkrete Algorithmus würde nicht terminieren. auch hier sind die weiteren Alternativen uninteressant.

Mit diesen Werten kann man den Entscheidungs-Algorithmus für abstraktes Matching folgendermaßen definieren:

$$
\begin{array}{lcl}
\mathrm{Match}_f(v,p) & = & \equiv, \text{ wenn } p \text{ Variable ist} \\
\mathrm{Match}_f(\top,p) & = & \subseteq, \text{ wenn } p \text{ keine Variable ist} \\
\mathrm{Match}_f(\perp,p) & = & \perp, \text{ wenn } p \text{ keine Variable ist} \\
\mathrm{Match}_f(C\ w_1\dots w_n, C'\ p_1\dots p_m) & = & \not\subseteq, \text{ wenn } C \neq C' \\
\mathrm{Match}_f(C\ w_1\dots w_n, C\ p_1\dots p_n) & = & \mathrm{Matchem}_f(\{\mathrm{Match}_f(w_i,p_i) | 1 \le i \le n\}) \\
\mathrm{Match}_f(\langle w_1\dots w_n\rangle, p) & = & \mathrm{MatchUnion}_f(\{\mathrm{Match}_f(w_i,p) | 1 \le i \le n\}) \\
\mathrm{Matchem}_f(m_1,\dots,m_n) & = & \equiv, \text{ wenn } (\forall 1 \le i \le n) : m_i = \equiv \\
 & = & \not\subseteq, \text{ wenn } (\exists 1 \le j \le n) : m_j = \not\subseteq \wedge \\
 & & \qquad\qquad (\forall 1 \le i < j) : m_i = \equiv \\
 & = & \perp, \text{ wenn } (\exists 1 \le j \le n) : m_j = \perp \wedge \\
 & & \qquad\qquad (\forall 1 \le i < j) : m_i = \equiv \\
 & = & \subseteq, \text{ sonst.} \\
\mathrm{MatchUnion}_f(m_1,\dots,m_n) & = & \equiv, \text{ wenn } (\forall 1 \le i \le n) : m_i = \equiv \\
 & = & \not\subseteq, \text{ wenn } (\forall 1 \le i \le n) : m_i = \not\subseteq \\
 & = & \perp, \text{ wenn } (\forall 1 \le i \le n) : m_i = \perp \\
 & = & \subseteq, \text{ sonst.}
\end{array}
$$

In $\mathrm{Matchem}_f$ und $\mathrm{MatchUnion}_f$ wird die funktionale Reduktionsstrategie verwendet. $\mathrm{Matchem}_f$ liefert nur dann $\not\subseteq$, wenn eines der Resultate $\not\subseteq$ ist und alle vorherigen $\equiv$ sind. Intuitiv matcht ein Konstruktor-Pattern einen Wert mit passendem Wurzelsymbol nur dann überhaupt nicht, wenn eines der Argument-Patterns seinen entsprechenden Wert überhaupt nicht matcht, aber alle vorher gematchten Werte keine andere Alternative zulassen. Entsprechendes gilt für $\perp$.

Da Vereinigungen eben Vereinigungen von Mengen konkreter Graphen sind führt die Forderung nach Sicherheit dazu, daß $\mathrm{MatchUnion}_f$ $\not\subseteq$ liefert, wenn alle Komponenten $\not\subseteq$ liefern, also alle Komponenten überhaupt nicht matchen. Hier gilt entsprechendes für $\perp$ und $\equiv$.

Operational bedeuten $\subseteq$ und $\not\subseteq$, daß weitere Alternativen in Betracht gezogen werden müssen, während bei $\perp$ und $\equiv$ keine weiteren Alternativen betrachtet werden müssen und $\perp$ bzw. $\equiv$ resultieren.

Einige Beispiele für das Pattern-Matching mit $\mathrm{Match}_f$ und dem Pattern

$$[\,] : (a : b)$$

sind:

$$
\begin{array}{lll}
\top : \top & \text{ergibt} & \subseteq, \\
[\,] : [\,] & \text{ergibt} & \not\subseteq, \\
\top : \perp & \text{ergibt} & \subseteq \text{ und} \\
[\,] : \perp & \text{ergibt} & \perp.
\end{array}
$$

Auch $\text{Match}_f$ kann noch verbessert werden. Beispiele für Fälle, in denen Information verloren geht, sind:

$$\begin{array}{lcl} \texttt{f [] (a:b)} & = & e_1 \\ \texttt{f x y} & = & e_2 \end{array}$$

Matching mit `f` $\top\ \bot$ ergibt dann $\subseteq$ auf der ersten Alternative und beide Alternativen werden ausgewählt. Benötigt wird jedoch nur die zweite, da Matching von $\bot$ mit $(a : b)$ in der ersten Alternative nicht terminieren würde.

Das bis jetzt Gesagte bezieht sich auf Pattern im allgemeinen, die aber im Sprachkern nicht verwendet werden. Dort werden nur simple Patterns verwendet und für die gestaltet sich der Algorithmus einfacher.

In simplen Patterns treten nur Konstruktor-Patterns auf, deren Argumente nur Variablen sein dürfen. Es folgt:

$$\text{Match}_f(C\ w_1\ \ldots\ w_n, C\ v_1\ \ldots\ v_n) = \equiv.$$

Wir haben immer simple Patterns, die also aus einem Konstruktor und entsprechend vielen Variablen bestehen. Dies wird den Algorithmus wesentlich vereinfachen, gegenüber dem, den Nöcker in [Nö93] verwendet. Wir überlassen also die Transformation komplexer Patterns, sollten sie in der Ausgangssprache vorhanden sein, in simple Patterns einer anderen Phase des Compilers. Daß dies die Ausdrucksstärke des Sprachkerns nicht mindert, haben wir in 2.2.3 beschrieben.

$$\begin{array}{rcl} \text{Match}^{\#}(\top, p) & = & \text{Wahr} \\ \text{Match}^{\#}(\bot, p) & = & \text{Falsch} \\ \text{Match}^{\#}(C v_1 \ldots v_n, C' p_1 \ldots p_m) & = & \text{Falsch, wenn } C \neq C' \\ \text{Match}^{\#}(C v_1 \ldots v_n, C p_1 \ldots p_m) & = & \text{Wahr} \\ \text{Match}^{\#}(\langle v_1, \ldots, v_n \rangle, p) & = & \exists 1 \leq i \leq n : \text{Match}^{\#}(v_i, p) \end{array}$$

## 3.3 Eine entscheidbare $\leq$-Relation

Für die Pfadanalyse und die Vereinfachung von Vereinigungen benötigen wir eine $\leq$-Relation auf den abstrakten Werten. Die $\leq_\alpha$-Relation, wie sie auf den abstrakten Werten definiert ist, ist unentscheidbar, ganau wie die Striktheits-Analyse selbst, denn:

$$\texttt{f}\ \bot = \bot \iff \texttt{f}\ \bot \leq_\alpha \bot$$

weil mit $\forall x \in \mathcal{AG} : \bot \leq_\alpha x$ aus der rechten Seite $\bot \leq_\alpha \texttt{f}\ \bot \leq_\alpha \bot$ wird.

Also müssen wir $\leq_\alpha$ approximieren. Eine einfache Approximation gibt einige Eigenschaften der $\leq_\alpha$-Relation wieder:

$$\begin{array}{rcl} \bot & \leq' & t \\ t & \leq' & \top \\ t & \leq' & t \end{array}$$

$$\begin{array}{rcl} \mathtt{f}\ x_1,\ldots,x_n & \leq' & \mathtt{f}\ y_1,\ldots,y_n, \forall 1 \leq i \leq n : x_i \leq_\alpha y_i \\ t & \leq' & \langle x_1,\ldots,x_n\rangle, \exists 1 \leq i \leq n : t \leq_\alpha x_i \end{array}$$

$\leq'$ approximiert $\leq_\alpha$ insofern, als $t \leq' \Rightarrow t \leq_\alpha s$.

Bei komplexen Formen der Striktheits-Analyse treten zyklische Werte auf, mit denen diese einfache Approximation nicht umgehen kann. Man erweitert daher die $\leq'$-Relation, indem man ein drittes Argument einführt, das Paare von Graphadressen enthält. Intuitiv stellem diese Paare Annahmen über die $\leq_\alpha$-Relation zwischen den Graphen deren Adressen ein solches Paar enthält dar. $\leq^\# (x, y, \{(t_1, t_2)\})$ steht also intuitiv für die $\leq'$ Relation zwischen $x$ und $y$ unter der Annahme, daß $t_1 \leq t_2$. Daraus ergibt sich:

$$\begin{array}{rl} \leq^\# (\bot, t, P) & \\ \leq^\# (t, \top, P) & \\ \leq^\# (t, t, P) & \\ \leq^\# (t, t', P), & \text{wenn } (t, t') \in P \\ \leq^\# (t@(fx_1 \ldots x_n), t'@(fy_1 \ldots y_n), P), & \text{wenn } \forall 1 \leq i \leq n : \leq^\# (x_i, y_i, (t, t') : P) \\ \leq^\# (t, t'@\langle x_1, \ldots, x_n\rangle, P), & \text{wenn } \exists 1 \leq i \leq n : \leq^\# (t, x_i, (t, t') : P \end{array}$$

Auch $\leq^\#$ approximiert $\leq_\alpha$. $\leq^\#$ kann mit zyklischen Werten umgehen und findet insbesondere sowohl

```
let a = ⊤:b; b =  ⊤:b in a ≤# let x = ⊤:x in x
```

als auch

```
let x = ⊤:x in x ≤# let a = ⊤:b; b = ⊤:b in a
```

Die $\leq^\#$-Relation wird von der Funktion `lessEqual` im Programm dargestellt.

## 3.4 Pfadanalyse

Sowohl Matching, als auch die abstrakte Reduktion können Nicht-Terminierung verursachen. Matching kann nicht-terminieren, wenn das Pattern oder der Wert einen Zyklus enthält. Im Sprachkern sind keine zyklischen Patterns gestattet also haben wir auch in den abstrakten Ausdrücken keine. Zyklen im Wert sind möglich, stören aber nicht, da ja in den Patterns keine auftreten können. Die einzige Gefahr stellen Zyklen dar, die mittels Vereinigungen entstehen. In 3.2.1 wurde gezeigt, daß einfache Vereinigungen nicht zyklisch sind. Wir verlangen also, daß die Vereinigungen vereinfacht sind bzgl. der auf ihnen definierten Äquivalenzen. Bleiben die nicht-terminierenden Reduktionen. Es ist glücklicherweise immer sicher einen Term, der nicht in WHNF ist, durch $\top$ zu ersetzen. Eine Implementation könnte dies aus verschiedenen Gründen tun.

1. die Laufzeit der Reduktion eines Terms übersteigt ein Maximum
2. der Speicherbedarf der Reduktion eines Terms übersteigt ein Maximum

Die Reduktion selbst muß zwar nicht terminieren, aber wir können Terminierung erzwingen, indem wir in einigen der obigen Fälle mit $\top$ approximieren. Dies ist jedoch nicht sehr genau, es geht dabei mehr Information verloren, als wir verlieren müßten. Betrachten wir dazu die Funktion `f x = f x`, dann hätten wir `f` $\perp \to$ `f` $\perp \to \ldots \to$ `f` $\perp \to \top$. Diese Folge abstrakter Reduktionen stellt eine Menge von Reduktionsfolgen in der konkreten Domäne dar, die nicht terminieren. Wir können also davon ausgehen, daß, wenn ein abstrakter Term reduziert wird zu einem Term, in dem er selbst an benötigter Stelle auftritt, die Auswertung der korrespondierenden konkreten Terme ebenfalls nicht terminiert.

In der **Regel für $\perp$-Einführung** wird dies formalisiert:

$$t \to_\alpha^* C(t) \wedge t \text{ wird benötigt} \Rightarrow t \to_\alpha \perp$$

Die $\perp$-Einführung ist ein abstrakter Reduktionsschritt . Für den Fall orthogonaler Termersetzungssysteme findet sich ein Beweis in [Nö92].

Man kann diese Regel verallgemeinern indem man nicht den Term selbst in einem Kontext $C$ erwartet, sondern einen, der höchstens die Terme in $t$ repräsentiert.

Die **Regel für allgemeine $\perp$-Einführung** lautet dann:

$$t \to_\alpha^* C(t') \wedge t' \leq_\alpha t \wedge t \text{ wird benötigt } \Rightarrow t \to_\alpha \perp$$

Aus $t \to_\alpha C(t') \ \wedge\ t' \leq_\alpha t \ \wedge\ t$ wird benötigt folgt $C(t') \to_\alpha C(C(t')) \ \wedge\ C(t')$ wird benötigt, denn es gilt $t \to_\alpha C(t') \iff (\forall g \in \gamma(t))(\exists h \in \gamma(C(t'))) : g \to_n h$. Wegen $t' \leq_\alpha t$ gilt $\gamma(t') \subseteq \gamma(t)$, also auch $t' \to_\alpha C(t')$ und da $t'$ in $C(t')$ benötigt wird, folgt $t \to_\alpha C(t') \to_\alpha C(C(t')) \to_\alpha \ldots$. Aus $C(t') \to_\alpha C(C(t'))$ ergibt sich $C(t') \to_\alpha \perp$ mit der Regel über $\perp$-Einführung.

Intuitiv bedeutet das, daß der aus $t$ reduzierte Ausdruck $C(t')$ weiterreduziert werden kann zu einem Ausdruck $C(C(t'))$, der an Stelle von $t'$ den zuerst aus $t$ reduzierten Ausdruck $C(t')$ enthält, denn es wird $t'$ in $C(t')$ benötigt und die Konkretisierungen von $t'$ sind in denen von $t$ enthalten, daher gilt auch $t' \to_\alpha C(t')$. Weiterhin wird dann $C(t')$ in $C(C(t'))$ benötigt und wir können so immer weiter fortfahren.

In unserer Implementation verwenden wir die allgemeine $\perp$-Einführung und werden im Folgenden auch die allgemeine $\perp$-Einführung einfach mit $\perp$-Einführung bezeichnen.

Wenn wir keine Information über die Notwendigkeit von $t'$ im abgeleiteten Ausdruck haben, dann können wir zwar nicht $\perp$ einführen, aber wir können einen zyklischen Wert einführen.

Die **Regel für Zyklus-Einführung** lautet also:

$$t \to_\alpha^* C(t') \wedge t' \leq_\alpha t \Rightarrow t \to_\alpha a@C(a).$$

## 3.5 Abhängigkeits-Analyse

Um den Resourcenverbrauch bei der Striktheits-Analyse, sowohl Zeit als auch Platz, gering zu halten, empfiehlt es sich, Striktheits-Information von Funktionen bereits zu kennen, wenn die Funktionen in einer anderen verwendet werden. Dadurch wird vermieden, daß die aufgerufene Funktion (abstrakt) reduziert werden muß, wenn eines der strikten Argumente $\perp$ ist.

Mit einer Abhängigkeits-Analyse kann man sicherstellen, daß die Funktionen in der gewünschten Reihenfolge analysiert werden. Die Abhängigkeits-Analyse verwendet den der Relation „$f$ ruft $g$ auf" entsprechenden Graphen, aus dem die starken Zusammenhangskomponenten, jeweils beim kleinsten Knoten (bzgl. obiger Relation) beginnend, nacheinander entfernt werden. Die Funktionen werden dann in der Reihenfolge analysiert, in der sie entfernt wurden. Im Anhang A finden sich Funktionen zur Abhängigkeits-Analyse. Bei einer Implementierung in einem vollständigen Compiler ist es unwahrscheinlich, daß die Striktheits-Analyse vorab eine Abhängigkeits-Analyse durführen muß, da diese Information bereits für die Typprüfung benötigt wird.

# Kapitel 4

# Umsetzung

In diesem Kapitel werden wir die Umsetzung der Methode aus dem vorangegangenen Kapitel darlegen. Dazu beschreiben wir zunächst die wesentlichen Merkmale, Überlegungen und Entscheidungen, auf denen unsere Umsetzung basiert, um schließlich mit der operationalen Semantik der $G^{\#}$-Maschine zu einer formaleren Darstellung unserer Umsetzung zu kommen.

## 4.1 Abstrakte Graphersetzung

Abstrakte Graphersetzung bzw. abstrakte Reduktion steht in direktem Bezug zu Graphersetzung bzw. Reduktion. Aufgrund dieser Verwandtschaft und dem Erfolg der G-Maschine bei der Umsetzung der Graphreduktion, erscheint es sinnvoll eine abstrakte Maschine für die abstrakte Graphersetzung auf Basis der G-Maschine zu entwerfen.

Unsere Maschine, die $G^{\#}$-Maschine, stellt die abstrakte Graphersetzung, die Pfadanalyse sowie $\perp$- und Zyklus-Einführung recht effizient dar. Eines der Ziele beim Entwurf der $G^{\#}$-Maschine war, die Ähnlichkeit zur G-Maschine aus [PJL91] weitgehend beizubehalten, nicht zuletzt, weil Teile dieser Arbeit als Ergänzung zu [PJL91] vorgesehen sind.

### 4.1.1 Abstrakte Werte

Die G-Maschine kann natürlich nicht mit den abstrakten Werten umgehen, die für die Umsetzung der abstrakte Graphersetzung benötigt werden. Für die Verarbeitung der abstrakten Werte bedarf es Änderungen gegenüber der G-Maschine in folgenden Bereichen:

- Anweisungen für primitive Funktionen
- `Casejump`-Anweisung

- `Split`-Anweisung
- `MkUnion`-Anweisung
- `Unwind`-Anweisung
- `AbsEq`-Anweisung

Betrachten wir diese Änderungen detaillierter.

### Anweisungen für primitive Funktionen

Die primitiven arithmetischen, vergleichenden und zeichenorientierten Anweisungen müssen verändert werden. Kann in der G-Maschine noch davon ausgegangen werden, daß das Programm typgerecht ist und Elemente eines passenden Typs an die Anweisungen übergeben werden, so muß in der $G^{\#}$-Maschine zwischen solchen Elementen und den abstrakten Werten unterschieden werden. Dies macht die primitiven Funktionen der $G^{\#}$-Maschine deutlich aufwendiger als die der G-Maschine. Die Vereinigungen stellen dabei den aufwendigsten Fall dar. Wir müssen nämlich für jedes Element der Vereinigung die primitive Anweisung ausführen, wobei das Element an der Stapelspitze steht. Weil diese primitiven Anweisungen den Heap verändern und wir schließlich einen Zustand benötigen, in dem alle Adressen, die die Funktionen als Stapelspitzen zurückgegeben haben, gültige Heapadressen sind, muß der Heap weitergereicht werden. Die Ergebnisse der primitiven Anweisungen werden dann eingesammelt und eine Vereinigung wird für sie angelegt.

Die Veränderungen für den abstrakten Wert $\top$ ist dagegen naheliegend, hier wird einfach $\top$ zurückgegeben.

Für $\perp$ brauchen wir keine Veränderung an den primitiven Anweisungen, wenn wir verlangen, daß die Argumente zu WHNF ausgewertet sind. Die Funktionen, die primitive Anweisungen verwenden, sind dann strikt in den entsprechenden Argumenten. Also wird mit $\perp$ nie wirklich eine primitive Anweisung ausgeführt, sondern es wird die Striktheits-Information für die Funktionen verwendet und wenn eines der Argumente $\perp$ ist, $\perp$ zurückgegeben.

### `Split`

`Split` dient dazu die Argumente aus einem Konstruktor herauszulösen und auf den Stapel zu legen. So wird das Binden der Werte an Pattern-Variablen dargestellt. `Split` kann nur als erste Anweisung einer Alternative in einer `Casejump`-Anweisung auftreten. Vereinigungen und $\perp$ werden bei `Casejump` abgefangen und besonders behandelt, daher kann `Split` von den abstrakten Werten nur $\top$ auf dem Stapel vorfinden, worauhin entsprechend viele $\top$s auf den Stapel gelegt werden müssen. Dies entspricht Pattern-Matching mit $\top$, wobei jede der Pattern-Variablen an $\top$ gebunden wird.

### `Casejump`

Ein weiterer Teil in dem sich $G^{\#}$-Maschine und G-Maschine unterscheiden ist die `Casejump`-Anweisung. Anhand des abstrakten Wertes auf dem Stapel müssen wir verschiedene Fälle betrachten. Der einfachste davon ist wohl $\bot$, dabei wird $\bot$ zurückgegeben.

Im Fall von $\top$ muß jede Alternative mit dem $\top$ auf dem Stapel ausgeführt werden. Dadurch entstehen möglicherweise mehrere Resultate, die dann in einer Vereinigung auf den Stapel gelegt werden. Außerdem wird in der Pfadkomponente des resultierenden Zustands der zurückgelegte Pfad für jede dieser Alternativen abgelegt. Dies dient ausschließlich der Anschaulichkeit und kann für Striktheits-Analyse im Alltag weggelassen werden.

Im Fall von Vereinigungen wird für jeden Wert der Vereinigung `Casejump` ausgeführt und das Resultat ist die Vereinigung der dabei entstehenden Resultate.

### `MkUnion`

Die `MkUnion` Anweisung ist in der G-Maschine nicht vorhanden. Mit ihr können mehrere Werte auf dem Stapel durch ihre Vereinigung ersetzt werden. Wir haben sie dem Instruktionsvorrat unserer Maschine hinzugefügt, damit der Compiler Code erzeugen kann, der explizit Vereinigungen erzeugt. Dadurch konnten die `<>`-Klammern im Sprachkern eingeführt werden, die zwar für die Striktheits-Analyse nicht notwendig sind, aber bei den Tests und der Fehlersuche überaus hilfreich waren.

### `Unwind`

Wohl die größten Unterschiede zur G-Maschine finden wir bei der `Unwind`-Anweisung der $G^{\#}$-Maschine. Davon bezieht sich jedoch nur ein Teil auf die Handhabung der abstrakten Werte. Die Umsetzung von $\bot$- und Zyklus-Einführung machen die restlichen Änderungen nötig.

Die Werte $\bot$ und $\top$ verhalten sich bei `Unwind`, wie andere Werte in (abstrakter) WHNF.

Vereinigungen können Redexe sein oder sie können in WHNF sein. Sind sie in WHNF, verfahren wir mit ihnen wie mit anderen Werten in WHNF. Sind es jedoch Redexe, ist jedes Element Redex und muß weiter reduziert werden. Dabei können wir prinzipiell eine von zwei Strategien verfolgen, nämlich zuerst in die Tiefe des Berechnungsgraphen zu gehen oder zuerst in die Breite zu gehen. Geht man zuerst in die Tiefe, reduziert also jedes Element zu WHNF bevor man mit dem nächsten Element fortfährt, ergeben sich einige Nachteile. Diese werden wir bei $\bot$- und Zyklus-Einführung besprechen, da sie sich in diesen Bereichen auswirken. Deshalb reduzieren wir in der $G^{\#}$-Maschine jede Komponente einen Schritt weiter und fassen sie dann zu einer Vereinigung zusammen, mit der wir die ursprüngliche Vereinigung auf dem Stapel ersetzen.

`AbsEq`

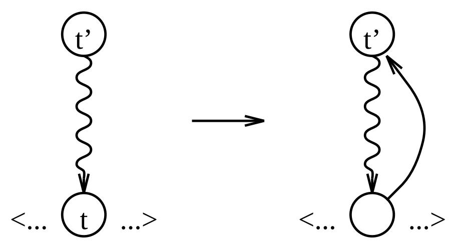
Die `AbsEq` Anweisung verwenden wir, um Knoten, im Gegensatz zu Werten in `Num` Knoten, miteinander vergleichen zu können. Damit kann unsere Maschine entscheiden, ob $\perp$ auf dem Stapel liegt oder nicht.

### 4.1.2 Pfadanalyse

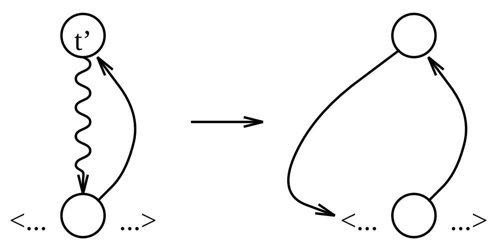
Bei der Umsetzung der Pfadanalyse wird diese in zwei Phasen unterteilt. In der ersten Phase werden die Kandidaten für $\perp$- und Zyklus-Einführung ermittelt und in der zweiten Phase wird entschieden welche der Einführungen tatsächlich vorzunehmen sind.

Ein Kandidat für eine dieser Einführungen ist ein Term $t$ in dessen Berechnungsgeschichte ein Term $t'$ auftritt, so daß gilt $t \leq t'$. Wir wissen noch nicht für welche, weil wir nicht wissen, ob $t$ benötigt wird, können aber zunächst einen Verweis einführen. Dazu überschreiben wir $t$ mit dem Verweis auf $t'$. Erst wenn eine Wurzel $w$ durch einen Wert $a$ überschrieben werden soll wird geprüft, welche der Einführungen wir vornehmen können. Enthält jeder Reduktionspfad, der durch $a$ dargestellt wird, einen Verweis auf $w$, können wir $\perp$ einführen, ansonsten wird durch überschreiben von $w$ mit einem Verweis auf $a$ Zyklus-Einführung vorgenommen.

Ein Beispiel soll dies noch deutlicher machen. Sei $t'$ reduziert zu $C(t)$, wobei der Kontext eine Vereinigung sein soll, also $C(t) = \langle \ldots, t, \ldots \rangle$. Sei weiter $t \leq^{\#} t'$, so daß wir $t$ durch einen Verweis auf $t'$ ersetzen.




Soll dann $t'$ überschrieben werden und nicht alle Komponenten der Vereinigung enthalten den Verweis auf $t'$, erfolgt Zyklus-Einführung durch Überschreiben von $t'$ mit dem Verweis auf die Vereinigung.




Die Änderungen, die für die Umsetzung der Pfadanalyse notwendig sind, beziehen sich auf:

- die Zustandskomponenten `GmOpen` und `GmMarks`

- den Typ der Heapzellen
- die Berücksichtigung der Marken in `update`
- den Beitrag von `update` zur Pfadanalyse durch `inEveryReductionPath`
- das Verhalten von `unwind` für `Ap`, `Ind` und `Union` Knoten

Auf diese Punkte werden wir nun im einzelnen eingehen.

### Zustandskomponenten `GmOpen` und `GmMarks`

Offensichtlich benötigen wir eine Methode auf die Terme der Berechnungsgeschichte zuzugreifen. Wir ermöglichen dies durch eine Zustandskomponente, `GmOpen`, in der Graphadressen gespeichert werden, von denen reduziert wird, d. h. die bei einem `Unwind` an der Stapelspitze liegen. Es ist dabei nicht nötig beliebige Graphadressen in `GmOpen` zu speichern, sondern es genügt die Adressen von Redexen zu speichern. Wenn nämlich ein Term nicht Redex ist, ist er mit Sicherheit auch nicht der Vorfahr eines anderen Terms in dessen Berechnungsgeschichte. Es müssen also nie Adressen von anderen Knoten (Zahlen, Konstruktoren, $\top$, $\perp$, globale Definitionen, Verweise) in `GmOpen` gespeichert werden, denn die können keine Redexe sein. Die einzige Art Knoten, die Redexe sein können, außer den Anwendungsknoten sind die Vereinigungen, aber für die wird die Berechnung gegebenenfalls in mehrere Pfade verzweigt, von denen dann jeder mit einem Anwendungsknoten auf dem Stapel beginnt, so daß auch die Adressen von Vereinigungen, die Redexe sind, nicht in `GmOpen` gespeichert werden müssen. Ebenso ist es nicht nötig die Adressen in `GmOpen` zu lassen, wenn die Wurzel des Graphen auf dessen Reduktion die Adresse zu `GmOpen` hinzugefügt wurde, durch einen Term in WHNF überschrieben wird, denn durch das Überschreiben endet die Berechnungsgeschichte des Teilgraphen und damit auch die aller seiner Teilgraphen. Daher können die während dieser Geschichte in `GmOpen` abgelegten Adressen keine Kandidaten mehr für eine der Einführungen sein.

Zusammenfassend können wir sagen, daß Adressen von Anwendungsknoten, die Redexe sind in `GmOpen` gespeichert werden, solange die Wurzel von der sie stammen noch nicht in WHNF ist. Wir nennen diese Redexe *offene Redexe*.

In unserer Umsetzung werden nur Anwendungsknoten, die Redexe sind, mit Graphen in `GmOpen` verglichen, um Kandidaten für $\perp$- und Zyklus-Einführung zu finden. Aus oben gesagtem folgt, daß dies ausreicht.

Tritt nun in einer Berechnungsgeschichte eine Vereinigung auf müssen mehrere alternative Berechnungen weiterverfolgt werden. Hierbei bietet sich ein Spektrum verschiedener Möglichkeiten. U. a. könnte man versuchen jede Komponente zu WHNF zu reduzieren, sofern sie eine hat oder man könnte auf jede Komponente ihre nächste Anweisung anwenden usw.

Vereinigungen entstehen häufig aufgrund rekursiver Funktionen. Würden wir dann z. B. immer die erste Komponente einer Vereinigung weiterreduzieren würden wir unter Umständen die Berechnung abbrechen müssen, ohne Striktheit gefunden zu haben. Gelangt nämlich eine rekursive Ersetzung immer in die erste Komponente und ist der Unterschied zu den früheren Anwendungen dieser Funktion so groß, daß unsere $\leq^{\#}$-Relation nicht zutrifft, obwohl eine $\leq_{\alpha}$-Relation zwischen den Termen besteht, dann erschöpfen wir irgendwann unsere Resourcen und approximieren mit $\top$.

Es ist aber durchaus möglich, sogar naheliegend, daß die anderen Ersetzungen, besonders die nicht-rekursiven, indem die Wurzel eines mehrfachverwendeten Teilgraphen überschrieben wird, dazu beitragen, daß nun die $\leq^{\#}$-Relation zwischen Termen besteht und wir Striktheit finden, die wir sonst nicht gefunden hätten. Ein einfaches Beispiel hierfür ist `length`:

```
length xs n  =  case xs of
                    <1> -> n;
                    <2> a b -> length b (n+1);
```

Wenn wir hier immer wieder die rekursive Alternative wählen und versuchen, so zu WHNF zu gelangen, geschieht folgendes:

$$\begin{array}{rcl}
\texttt{length}\ \top\ \bot & \rightarrow & \langle \bot, \texttt{length}\ \top\ (\bot\texttt{+1}) \rangle \\
\texttt{length}\ \top\ (\bot\texttt{+1}) & \rightarrow & \langle \bot\texttt{+1}, \texttt{length}\ \top\ ((\bot\texttt{+1})\texttt{+1}) \rangle \\
\texttt{length}\ \top\ ((\bot\texttt{+1})\texttt{+1}) & \rightarrow & \langle (\bot\texttt{+1})\texttt{+1}, \texttt{length}\ \top\ (((\bot\texttt{+1})\texttt{+1})\texttt{+1}) \rangle \\
 & \vdots &
\end{array}$$

Wir könnten unseren Reduktionsalgorithmus so verändern, daß immer zuerst eine nicht-rekursive Alternative weiterreduziert wird. Das Problem dabei ist, daß es nicht unbedingt mit einfachen Mitteln erkennbar sein muß, welches rekursive bzw. nicht-rekursive Alternativen sind.

Was wir jedoch tun können, ist jede Komponente ein Stück weiter zu reduzieren und die jeweiligen, etwas weiter reduzierten, Graphen dann zu vereinigen. Nun gehört zu jeder Komponente einer Vereinigung ein Berechnungspfad mit seinen offenen Redexen, also speichern wir die offenen Redexe bei jeder Komponente einer Vereinigung. Um eine Komponente ein Stück[1] weiter zu reduzieren, gehen wir so vor. Wenn eine Komponente weiter reduziert werden soll, dann markieren wir die Wurzel des entsprechenden Graphen, so daß wir beim Überschreiben einer Wurzel nachsehen können, ob sie markiert war. Diese Marken befinden sich in der Zustandskomponente `GmMarks`, in Form der Adresse der Wurzel. War eine Wurzel markiert, so wollen wir, sobald sie überschrieben ist, mit einer anderen Komponente unserer Vereinigung weitermachen. Dazu wird in die Anweisungsfolge eine `Join` Anweisung eingefügt, mit dem Effekt, daß die Auswertung aufhört und eine andere Komponente ausgewertet werden kann.

Ein zusätzlicher Vorteil dieser Methode ist, daß nachdem alle Komponenten einen Schritt weiter reduziert wurden und wieder vereinigt werden, eine Vereinfachung

[1]Ein Stück ist hier eine abstrakte Reduktion. Wir könnten aber auch mehrere abstrakte Reduktionen weiterreduzieren oder nur wenige $G^{\#}$ Maschinen Anweisungen.

der entstehenden Vereinigung stattfinden kann, wodurch der zu untersuchende Berechnungsbaum verkleinert wird.

Zusammenfassend können wir sagen, daß Vereinigungen komponentenweise jeweils einen Schritt weiterreduziert werden, wozu `GmMarks` und die bei den Komponenten gespeicherten offenen Redexe verwendet werden.

Wenden wir uns nun einem weiteren Problem zu, das sich aus der Tatsache ergibt, daß die Graphen im Heap, deren Adressen in `GmOpen` gespeichert sind, weiterreduziert werden können. Durch solche Reduktion könnte es vorkommen, daß die $\leq^{\#}$-Relation nicht mehr besteht, obwohl sie vor der Reduktion bestand. Auch hier verwenden wir das `length` Beispiel.

$$\begin{aligned}
\texttt{length}\ \top\ \bot \ &\rightarrow\ \langle \texttt{length}\ \top\ b@(\bot+1), \bot\rangle \\
&\qquad \equiv_\alpha \texttt{length}\ \top\ b@(\bot+1) \\
&\rightarrow\ \langle \texttt{length}\ \top\ (b@(\bot+1)+1), b@(\bot+1)\rangle \qquad (4.1)\\
&\rightarrow\ \langle \texttt{length}\ \top\ ((b@\bot+1)+1), b@\bot+1\rangle
\end{aligned}$$

Das sieht zunächst vielversprechend aus, da aber mittlerweile $b@(\bot+1)$ zu $b@\bot$ reduziert wurde, steht in 4.1 jetzt $\langle \texttt{length}\ \top\ (b@\bot+1), b@\bot\rangle \equiv_\alpha \texttt{length}\ \top\ (b@\bot+1)$ und für die Pfadanalyse müßten wir entscheiden, ob $\langle \texttt{length}\ \top\ ((\bot+1)+1),\ \bot+1\rangle \leq^{\#} \texttt{length}\ \top\ (\bot+1)$. Aus der Definition von $\leq^{\#}$ ergibt sich:

$$\begin{aligned}
&\langle \texttt{length}\ \top\ ((\bot+1)+1), \bot+1\rangle \leq^{\#} \texttt{length}\ \top\ (\bot+1) \\
&\quad\Longleftrightarrow\ \bot+1 \leq^{\#} \bot\ \wedge\ \bot+1 \leq^{\#} \texttt{length}\ \top\ (\bot+1)
\end{aligned}$$

Beide Teile der Konjunktion sind falsch, tatsächlich gilt aber $\bot+1 \leq_\alpha \bot\ \wedge\ \bot+1 \leq_\alpha \texttt{length}\ \top\ (\bot+1)$.

Da $\bot+1 \leq_\alpha \bot$ nur dann gilt, wenn $\bot+1 = \bot$, finden wir uns sogar beim Ausgangsproblem, der Striktheits-Analyse, wieder. Der zweite Teil der Konjunktion ist mindestens so schwer wie der erste.

Für die Pfadanalyse ist es also wichtig, die offenen Redexe zu speichern, bevor Reduktion stattfindet.

So bleibt uns im Beispiel die Zeile 4.1 für die Pfadanalyse erhalten und wir müssen auswerten:

$$\langle \texttt{length}\ \top\ ((\bot+1)+1), \bot+1\rangle \leq^{\#} \langle \texttt{length}\ \top\ ((\bot+1)+1), \bot+1\rangle$$

was wahr ergibt.

Im folgenden Abschnitt beschreiben wir, wie in unserer Umsetzung die offenen Redexe gespeichert werden.

### Heapstruktur mit Paaren von Knoten

Jede Zelle im Heap besteht tatsächlich aus zwei Knoten, der erste davon ist der Wurzelknoten eines ursprünglichen Teilgraphen, während der zweite der Wurzelknoten des möglicherweise bereits reduzierten Graphen ist. So steht der ursprüngliche Redex zur Verfügung, ohne daß kopiert werden muß. Vom Platzbedarf her sind Kopieren und Knotenpaare Verwenden äquivalent. Vom Zeitbedarf her hat Kopieren den Nachteil, daß darauf geachtet werden muß, daß die Graphstruktur mit ihren möglichen Mehrfachverwendungen getreu kopiert wird.

### `update` und Pfadanalyse

Ein Teil der Pfadanalyse wird von der `update`-Funktion realisiert. Zunächst unterscheidet `update` zwei Arten von Wurzeln: markierte und nicht markierte. `unwind` markiert Wurzeln, wenn alternative Ausdrücke zur Auswertung vorliegen, wie bei Vereinigungen. `update` fügt beim Überschreiben markierter Wurzeln eine `Join` Anweisung in den Code ein, um die Auswertung für den Ausdruck zu beenden. Außerdem wird die Marke der Wurzel entfernt. Sowohl markierte als auch nicht markierte Wurzeln entfernt `update` aus der Liste der offenen Redexe, `GmOpen`, und ebenso alle Verweise bzw. Verweisketten, die zur überschriebenen Wurzel führen.

`update` ist auch die Funktion, die `inEveryReductionPath` aufruft, in der entschieden wird, ob nur ein Zyklus einzuführen ist oder ob $\perp$ eingeführt werden kann. Findet `inEveryReductionPath` in jedem Ausdruck, den die zu überschreibende Wurzel repräsentiert einen Verweis auf diese Wurzel, kann $\perp$ eingeführt werden.

Für die verschiedenen Arten von Knoten gelten dabei folgende Regeln:

- Verweise werden verfolgt, es sei denn sie verweisen auf die Wurzel.
- Bei Anwendungen genügt es, wenn der Verweis auf die Wurzel in der Funktion oder im Argument auftritt.
- Auch genügt es, wenn der Verweis in einem Argument eines Konstruktors auftritt.

In diesen Fällen handelt es sich immer um die Darstellung *eines* Wertes bzw. Reduktionspfades. Vereinigungen stellen mehrere Reduktionspfade dar.

- In Vereinigungen muß der Verweis in jedem Element auftreten, denn hierbei werden mehrere alternative Werte repräsentiert.
- In anderen Knoten kann der Verweis nicht auftreten.

## 4.2 Notwendigkeit

In 3.4 haben wir gesehen, daß eine wesentliche Eigenschaft für die Definition der $\perp$-Einführung die Notwendigkeit ist. Hier werden wir besprechen, wie sich Notwendigkeit elegant umsetzen läßt.

Aus Theorem 3.2 wissen wir, daß bei Normalordnung nur notwendige Reduktionen durchgeführt werden. Erinnern wir uns an die Definitionen von allgemeiner $\perp$- und Zyklus-Einführung aus 3: der Unterschied zwischen beiden war, daß im einen Fall ein notwendiger Term vorlag, im anderen Fall ein nicht-notwendiger. Wir weichen nur bei den Vereinigungen möglicherweise von Normalordnung ab, so daß wir immer notwendige Reduktionen durchführen, außer möglicherweise bei Vereinigungen. Ein Term wird für eine Vereinigung benötigt, wenn er in der Auswertung jeder Komponente benötigt wird. Dann wird offensichtlich, warum wir Pfadanalyse in zwei Phasen aufteilen: erst nachdem die Kandidaten ermittelt sind können wir, indem wir untersuchen, ob ein Kandidat in allen Komponenten auftritt entscheiden, welche der Einführungsregeln angewandt werden kann.

Die Notwendigkeit eines Terms $t$ stellt, sozusagen posthum, nachdem $t$ durch einen Verweis auf einen offenen Redex ersetzt wurde, die Funktion `inEveryReductionPath` (siehe 4.1.2) fest, indem sie untersucht, ob der Verweis in jedem Reduktionspfad auftritt.

## 4.3 Speichern und Verwenden von Striktheits-Information

Ein weiterer wichtiger Bereich betrifft das Speichern und Verwenden der gefundenen Striktheits-Information, was uns in diesem Abschnitt beschäftigen soll.

Striktheits-Information über eine Funktion $f$ zu haben bedeutet zu wissen, welche ihrer Argumente $f$ zur Auswertung auf jeden Fall benötigt. Wie verwenden wir diese Information innerhalb der Striktheits-Analyse selbst? Wenn eine Anwendung einer Funktion auszuwerten ist möchten wir aufgrund schon vorhandener Striktheits-Information prüfen, ob eine der strikten Positionen mit $\perp$ besetzt ist und das ist die einzige Verwendung für Striktheits-Information, die wir in der Striktheits-Analyse haben. Auch hier läßt sich die Idee der G-Maschine verwenden: wir können nämlich für das Prüfen auf $\perp$ der strikten Positionen $G^{\#}$-Code erzeugen, der dann vor dem $G^{\#}$-Code der eigentlichen Funktion ausgeführt wird.

Haben wir also Striktheits-Information zu einer Funktion `f` ermittelt, dann speichern wir diese, indem wir:

1. `f` umbenennen in `_f`
2. eine Funktion `f` anlegen, die
   - für die strikten Argumente auf $\perp$ prüft
   - `_f` nur aufruft, wenn an keiner strikten Stelle $\perp$ steht

- $\perp$ liefert, wenn an ein striktes Argument $\perp$ ergibt

Für die Überprüfung der Argumente muß die $G^{\#}$-Maschine Gleichheit mit $\perp$ überprüfen können. Dazu ist die `Eq` Anweisung nicht in der Lage, denn `Eq` soll die Semantik des konkreten `==` repräsentieren. D. h. `Eq` muß strikt in beiden Argumenten sein. Offensichtlich kann `Eq` keine Information über Gleichheit auf dem abstrakten Wert $\perp$ liefern. Dazu führen wir eine Anweisung `AbsEq` ein, für die zumindest $\perp$ ‘`AbsEq`‘ $\perp$ gilt. Der $G^{\#}$-Code der für eine $n$-stellige Funktion `f` mit Striktheit $s = (s_1, \ldots, s_n)$ ist dann `analyserCode 0 s f`, wobei

```
> analyserCode :: Int -> [Bool] -> Name -> GmCode
> analyserCode arg [] f =   [Pushglobal (´_´:f)] ++
>           take arg (repeat Mkap) ++
>           [Update 0, Pop 0, Unwind]
> analyserCode arg (False:xs) f = analyserCode (arg+1) xs f
> analyserCode arg (True:xs) f  =
>   [Push arg,
>    Eval,
>    Pushglobal "Bot",
>    AbsEq,
>    Casejump [(2:= [Pushglobal "Bot",
>                    Update (arg+2+length xs),
>                    Pop (arg+2+length xs),
>                    Unwind]),
>              (1:= ((Pop 1):(analyserCode (arg+1) xs f)))]]
```

Hierbei steht die `2` in der `Casejump` Anweisung für „Wahr“, also für den Fall, daß $\perp$ vorliegt. Wird nun eine Funktion analysiert, die eine Anwendung von `f` enthält, dann nutzen wir die Striktheits-Information, indem dieser $G^{\#}$-Code ausgeführt wird.

## 4.4 Nicht-Strikte Umsetzung

Während in [Nö92] eine strikte Auswertung für notwendig gehalten wird, halten wir dies nicht für nötig. Das dort angegebene Beispiel macht diese Vorgehensweise jedenfalls nicht notwendig, auch werden wir informal erklären warum es nicht plausibel erscheint, daß eine solche Vorgehensweise nötig sein sollte.

Nöcker argumentiert, daß es oft besser ist eine strikte Reduktionsstrategie zu verfolgen, weil sonst bestimmte Fälle von Striktheit nicht gefunden werden. Um einen solchen Fall anzuführen verwendet er das beliebte `length`-Beispiel und stellt fest, daß die Striktheit im zweiten Argument nicht gefunden wird. Er betrachtet die Reduktion:

$$\begin{aligned} \texttt{length}\ \top\ \perp &\to_\alpha \langle \texttt{length}\ \top\ (\perp + 1), \perp\rangle \\ &\equiv_\alpha \texttt{length}\ \top\ (\perp + 1) \end{aligned}$$

Hier sieht er es als notwendig an zunächst $\perp + 1$ zu reduzieren, um dann mit der Pfadanalyse $\perp$ einführen zu können. Wie wir in 4.1.2 gezeigt haben, ist dies in unserer Umsetzung nicht nötig.

Warum erscheint die strikte Vorgehensweise generell nicht notwendig?

Die strikte Auswertung war im zitierten Beispiel ein Vorteil, weil dadurch $\perp$-Einführung möglich wurde. Ist die Auswertung eines Argumentes in einem der entstehenden Reduktionspfade notwendig, dann wird sie in unserer Umsetzung auch durchgeführt, und wir können ebenfalls $\perp$ einführen. Ist sie dagegen in keinem notwendig, würde sie in unserer Umsetzung nicht durchgeführt werden.

Möglicherweise kann ein Beispiel gefunden werden, in dem

- eine strikte Position untersucht wird,
- an einer anderen Position ein Term $t$ steht, der in keinem entstehenden Reduktionspfad benötigt wird und
- nur Reduktion von $t$ $\perp$-Einführung ermöglicht.

Wir konnten kein solches finden.

## 4.5 Verzicht auf Terminierungsanalyse

Nach unserer Definition der Striktheit sind auch Funktionen $f$ strikt, die unter keinen Umständen terminieren, denn dann gilt auch $f\ \perp = \perp$. Ein Beispiel für eine solche Funktion ist `g`:

```
g x  =  Cons{2,2} 1 (g x)
```

Die verwendete Methode kann dies in vielen Fällen herausfinden, dennoch wird in der Implementation nicht nach solchen Funktionen gesucht, da sie eher einen degenerierten Fall darstellen und daher den deutlich höheren Zeitaufwand (in Tests fast 50%) nicht rechtfertigen. In einem Compiler könnte man, beispielsweise durch Direktiven, Superkombinatoren angeben, für die Terminierung analysiert werden soll.

## 4.6 Operationale Semantik der $G^{\#}$-Maschine

Wie schon die G-Maschine, ist auch die $G^{\#}$-Maschine ein Zustandsüberführungssystem, wobei hier der Zustand ein 10-Tupel ist:

$$\langle \omega, C, S, D, H, G, \sigma, \pi, O, M \rangle$$

Die Großbuchstaben bezeichnen Komponenten, die für die Funktion der $G^{\#}$-Maschine unerläßlich sind, während die griechischen Buchstaben Komponenten bezeichnen, die Information enthalten, um die internen Abläufe anschaulicher zu machen. Es steht:

$\omega$ für die Ausgabe
$C$ für die Anweisungsfolge
$S$ für den Stapel
$D$ für den Dump
$H$ für den Heap
$G$ für die globalen Namen
$\sigma$ für statistische Informationen
$\pi$ für Pfade
$O$ für offene Redexe
$M$ für Marken

Die Bedeutung, die den einzelnen Komponenten zukommt ergibt sich aus den Zustandsübergängen für die Instruktionen. Der Instruktionsvorrat der $G^{\#}$-Maschine besteht aus:

`Eval`,
`Push` $n$, `Pushglobal` $g$, `Pushint` $i$,
`Mkap`, `MkUnion` $n$,`Pack` $t$ $n$,
`Pop` $n$, `Slide` $n$,
`Add`, `Sub`, `Mul`, `Div`, `Neg`,
`Le`, `Ge` `Lt`, `Gt`, `Eq`, `Ne`,
`Ord`, `Chr`,
`Casejump` $as$,
`Print`,
`Alloc` $n$,
`AbsEq`,
`Join`,
`Update` $n$,
`Unwind`

Der Graph wird durch den Heap dargestellt. Folgende Arten von Knoten stehen dazu zur Verfügung:

`Num` $n$ die Zahl $n$.

`Ap` $a_1$ $a_2$ die Anwendung der Funktion auf die $a_1$ zeigt auf das Argument auf das $a_2$ zeigt.

`Global` $s$ $c$ die Anweisungen $c$ für einen Superkombinator der Stelligkeit $s$.

`Ind` $a$ ein Verweis auf einen anderen Knoten.

`Constr` $r$ $as$ der $r$-te Konstruktor eines Typs mit der Liste der Adressen seiner Argumente.

`Top` der abstrakte Wert $\top^{\#}$.

`Bot` der abstrakte Wert $\bot^{\#}$.

`Union` $us$ die Vereinigung der abstrakten Graphen in $us$. $us$ ist eine Liste von Paaren, deren erste Komponente jeweils den zu vereinigenden Graphen bezeichnet und deren zweite Komponente die offenen Redexe auf dem dazugehörigen Reduktionspfad speichert.

`Visited` ein bereits besuchter Knoten. (für die Abhängigkeits-Analyse)

Da in den Vereinigunen nun auch die Liste der offenen Redexe zu jeder Komponente gespeichert wird, erweitern wir die Definitionen von $\setminus$ für Vereinigungen und Maxima von Vereinigungen in kanonischer Weise.

**Definition 4.1** *Sei*

$$\langle (x_1, o_1), \ldots, (x_n, o_n) \rangle \setminus x := \langle (x_{k_1}, o_{k_1}), \ldots, (x_{k_s}, o_{k_s}) \rangle,$$

*so daß* $\{x_{k_1}, \ldots, x_{k_s}\} = \{x_1, \ldots, x_n\} \setminus x$

**Definition 4.2** *Sei*

$$M(\langle (x_1, o_1), \ldots, (x_n, o_n) \rangle) := \begin{cases} \langle (x_{m_1}, o_{m_1}), \ldots, (x_{m_r}, o_{m_r}) \rangle & r \geq 2 \\ (x_{m_r}, o_{m_r}) & r = 1 \end{cases}$$

*mit* $(\forall 1 \leq i \leq r) : x_{m_i} \leq_\alpha x_{m_j} \Rightarrow i = j$.

Die entsprechende Definition der einfachen Vereinigungen erhalten wir, indem wir die neuen Definitionen von $\setminus$ und $M(\cdot)$ in Definition 3.9 einsetzen.

Keine der Anweisungen ändert alle Komponenten des Zustands auf einmal. Ist ein Zustandsübergang unabhängig von einer bestimmten Komponente, so wird diese im Ausgangszustand mit einem $*$ gekennzeichnet, wenn im resultierenden Zustand ein $*$ steht, deutet dieser an, daß die Komponente unverändert geblieben ist. Zur besseren Übersicht schreiben wir die Anweisung vor die Transition, also statt

$$\langle *, \texttt{Push 0} : C, s : S, *, \ldots, * \rangle \vdash \langle *, C, s : s : S, *, \ldots, * \rangle$$

schreiben wir

$$\texttt{Push 0} : \langle *, *, s : S, *, \ldots, * \rangle \vdash \langle *, *, s : s : S, *, \ldots, * \rangle$$

Die Endzustände für die $G^{\#}$-Maschine sind Zustände in denen keine Anweisungen mehr abzuarbeiten sind und Zustände, die die zur Verfügung gestellten Resourcen erschöpfen. Außerdem sind Zustände Endzustände, die einem Zustand folgen, dessen Anweisung die `Join` Anweisung ist.

Die Funktion *eval* bildet einen Zustand auf einen Endzustand ab, indem wiederholt Zustandsübergänge durchgeführt werden, bis ein Endzustand erreicht ist. Die Zustandsübergänge werden nach folgenden Regeln durchgeführt.

### 4.6.1 `Eval`

Die `Eval` Instruktion beginnt eine neue Auswertung des Graphen dessen Wurzel auf der Stapelspitze liegt, sofern es sich nicht um $\top$ oder $\bot$ handelt. Die Werte $\top$ und $\bot$ befinden sich in *abstrakter* WHNF, so daß hier keine Ersetzung stattfinden soll.

$$\texttt{Eval} : \langle *, C, a : S, D, H\left[a = \left\{ \begin{array}{l} \texttt{Top} \\ \texttt{Bot} \end{array} \right.\right], *, \ldots, *\rangle \vdash \langle *, \ldots, *\rangle$$

Ansonsten muß weitere Auswertung veranlaßt werden.

$$\begin{array}{rl} \texttt{Eval} : & \langle *, C, a : S, D, H\left[a \neq \left\{ \begin{array}{l} \texttt{Top} \\ \texttt{Bot} \end{array} \right.\right], *, \ldots, *\rangle \\ \vdash & \langle *, [\texttt{Unwind}], a, (C, S) : D, *, \ldots, *\rangle, \end{array}$$

`Eval` merkt sich also den Kontext, bestehend aus Anweisung $C$ und Stapel $S$, in dem die neue Auswertung begonnen wurde, auf dem Dump.

### 4.6.2 `Push`-Anweisungen

Die `Push`-Anweisungen legen Zeiger auf den Stapel, ohne im Heap Veränderungen vorzunehmen. `Pushint` macht dabei in einigen Umsetzungen eine Ausnahme[2], so auch hier.

$$\texttt{Push}\ n : \langle *, *, a_0 : \ldots : a_n : S, *, \ldots, *\rangle \vdash \langle *, *, a_n : a_0 : \ldots : a_n : S, *, \ldots, *\rangle$$

`Push` kopiert also nur den $n$-ten Stapeleintrag an die Spitze. Analog verfährt `Pushglobal`:

$$\texttt{Pushglobal}\ g : \langle *, *, S, *, *, G[g = a], *, \ldots, *\rangle \vdash \langle *, *, a : S, *, \ldots, *\rangle$$

Für `Pushint` unterscheiden wir den Fall, daß kein Wert auf dem Heap angelegt werden muß

$$\texttt{Pushint}\ i : \langle *, *, S, *, *, G['i' = a], *, \ldots, *\rangle \vdash \langle *, *, a : S, *, *, *, \ldots, *\rangle$$

von dem Fall, daß ein Wert auf dem Heap angelegt werden muß

$$\begin{array}{l} \texttt{Pushint}\ i : \langle *, *, S, *, H, G, *, \ldots, *\rangle \\ \quad \vdash \langle *, *, a : S, *, H', G', *, \ldots, *\rangle, \text{wobei } H' = H[a = \texttt{Num}\ i], G' = G['i' = a] \end{array}$$

Dabei ist $'i'$ ein kanonischer Name für $i$.

[2] In Umsetzungen, die einen eigenen Stapel für primitive Werte verwenden, statt diese als Knoten auf dem Heap anzulegen, macht `Pushint` keine Ausnahme.

### 4.6.3 Konstruktion von Knoten

Die Anweisungen `Mkap`, `MkUnion` $n$ und `Pack` $t$ $n$ haben die Aufgabe neue Knoten im Heap anzulegen.

`Mkap` nimmt die beiden oberen Stapeleinträge und macht aus ihnen einen Anwendungsknoten.

$$\texttt{Mkap} : \langle *, *, a_1 : a_2 : S, *, H, *, \ldots, * \rangle \vdash \langle *, *, a : S, *, H[a = \texttt{Ap}\ a_1\ a_2], *, \ldots, * \rangle$$

`MkUnion` ist für die Striktheits-Analyse nicht notwendig: sie macht es möglich die abstrakte Reduktion für Vereinigungen zu betrachten und dient damit der Anschauung. Dazu steht im Sprachkern das syntaktische Konstrukt der `<>`-Klammern zur Verfügung. `MkUnion` $n$ ersetzt $n$ Stapelelemente durch ihre Vereinigung.

$$\begin{aligned} &\texttt{MkUnion}\ n : \langle *, *, a_1 : \ldots : a_n : S, *, H, *, \ldots, * \rangle \\ &\quad \vdash \langle *, *, a : S, *, H[a = \texttt{Union}\ [a_1, \ldots, a_n]], *, \ldots, * \rangle \end{aligned}$$

`Pack` $r$ $n$ packt die $n$ oberen Stapeleinträge in den $r$-ten Konstruktor eines Typs.

$$\begin{aligned} &\texttt{Pack}\ t\ n : \langle *, *, a_1 : \ldots : a_n : S, *, H, *, \ldots, * \rangle \\ &\quad \vdash \langle *, *, a : S, *, H[a = \texttt{Constr}\ r\ [a_1, \ldots, a_n]], *, \ldots, * \rangle \end{aligned}$$

### 4.6.4 `Pop`-Anweisungen

Die `Pop` und `Slide` Anweisungen entfernen Einträge von der Stapelspitze. `Pop` entfernt die $n$ oberen Einträge.

$$\texttt{Pop}\ n : \langle *, *, a_1 : \ldots : a_n : S, *, \ldots, * \rangle \vdash \langle *, *, S, *, \ldots, * \rangle$$

`Slide` $n$ entfernt $n$ Einträge unter der Stapelspitze.

$$\texttt{Slide}\ n : \langle *, *, a : a_1 : \ldots : a_n : S, *, \ldots, * \rangle \vdash \langle *, *, a : S, *, \ldots, * \rangle$$

### 4.6.5 Anweisungen für primitive Funktionen

Die arithmetischen, die vergleichenden und die zeichenorientierten Anweisungen müssen $\top$ an die Stapelspitze legen, wenn eines ihrer Argumente $\top$ ist.

$$\begin{aligned} &\iota : \langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{Top}], *, \ldots, * \rangle \\ &\quad \vdash \langle *, *, a_1 : S, *, \ldots, * \rangle, \\ &\quad \forall \iota \in \{\texttt{Add}, \texttt{Sub}, \texttt{Mul}, \texttt{Div}, \texttt{Lt}, \texttt{Le}, \texttt{Gt}, \texttt{Ge}, \texttt{Eq}, \texttt{Ne}\} \end{aligned}$$

und entsprechend für $a_2$.

Auch für die `Neg`, `Ord` und `Chr` Anweisungen gilt entsprechendes, wobei aber nur der oberste Stapeleintrag verwendet wird. Da dann das eine Argument an der Stapelspitze gleich dem Resultat ist, muß keine Änderung am Zustand erfolgen.

$$\begin{array}{l} \iota : \langle *, *, a : S, *, H[a = \texttt{Top}], *, \ldots, * \rangle \quad \vdash \quad \langle *, \ldots, * \rangle \\ \forall \iota \in \{\texttt{Neg}, \texttt{Ord}, \texttt{Chr}\} \end{array}$$

Die primitiven Funktionen (die einzigen Stellen, in denen solche Anweisungen auftreten) sind alle strikt in ihren Argumenten, was garantiert, daß nie eines der Argumente für solche Anweisungen $\perp$ ist, denn für die strikten Argumente vergleichen wir mit $\perp$ und geben, wenn $\perp$ vorliegt, $\perp$ zurück.

Für Vereinigungen entstehen (möglicherweise) wieder Vereinigungen. Da wir fordern, daß Vereinigungen vereinfacht sind, definieren wir zunächst eine Funktion $v$, die diesen Teil der Zustandsüberführungsregeln darstellen kann.

$$\begin{array}{l} v(\langle *, *, S, *, H, *, *, *, O, * \rangle, (u_1, O_1), \ldots, (u_n, O_n)) \\ \quad = \quad \langle *, *, u' : S, *, H', *, *, *, O', * \rangle, \\ \text{wobei } m = M(x@\texttt{<}(u_1, O_1), \ldots, (u_n, O_n)\texttt{>} \setminus x) \text{ in} \\ \qquad \left.\begin{array}{lcl} u' & = & u, \\ H' & = & H, \\ O' & = & O^*, \end{array}\right\} \text{wenn } m = (u, O^*) \\ \qquad \left.\begin{array}{lcl} H' & = & H[u' = m], \\ O' & = & O, \end{array}\right\} \text{sonst} \end{array}$$

Betrachten wir zunächst die zwei-stelligen Anweisungen.

$$\begin{array}{l} \iota : \langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{<}(u_1, O_1), \ldots, (u_n, O_n)\texttt{>}], *, *, *, O, * \rangle \\ \quad \vdash v(\langle *, *, S, *, H^n, *, *, *, O, * \rangle, (u_1', O_1'), \ldots, (u_n', O_n')) \\ \quad \forall (\iota, \odot) \in \{(\texttt{Add}, +), (\texttt{Sub}, -), (\texttt{Mul}, *), (\texttt{Div}, \setminus)\} \\ \quad \text{dabei ist} \\ \quad \iota : \overbrace{\langle *, *, u_i : a_2 : S, *, H^{i-1}, *, *, *, O_i, * \rangle}^{s^{i-1}} \\ \quad \vdash \underbrace{\langle *, *, u_i' : S, *, H^i, *, *, *, O_i', * \rangle}_{s^i} \end{array}$$

Ganz analog wird für $a_2$ verfahren. Auch für `Neg`, `Ord` und `Chr` gelten entsprechende Regeln.

Ansonsten können bei allen dieser Anweisungen nur `Num` Knoten auf dem Stapel liegen, außer bei `Ord`, wo ansonsten nur ein `Constr` $t$ `[]` Knoten auf dem Stapel liegen kann. Wir unterscheiden arithmetische, vergleichende und zeichenorientierte Anweisungen.

### Arithmetische Anweisungen

Arithmetische Anweisungen führen die entsprechende Operation auf den in den Knoten dargestellten Zahlen aus und legen einen `Num` Knoten mit dem Resultat auf den Stapel.

$$\begin{aligned}\iota : &\langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{Num}\ x; a_2 = \texttt{Num}\ y], *, \ldots, * \rangle\\ &\vdash \langle *, *, a : S, *, H[a = \texttt{Num}\ x \odot y], *, \ldots, * \rangle,\\ &\forall(\iota, \odot) \in \{(\texttt{Add}, +), (\texttt{Sub}, -), (\texttt{Mul}, *), (\texttt{Div}, \backslash)\}\end{aligned}$$

`Neg` wird ganz entsprechend, nur eben mit einem Eintrag von der Stapelspitze, dargestellt.

$$\begin{aligned}\texttt{Neg} : &\langle *, *, a_1 : S, *, H[a_1 = \texttt{Num}\ n], *, \ldots, * \rangle\\ \vdash\ &\langle *, *, a : S, *, H[a = \texttt{Num}\ -n], *, \ldots, * \rangle\end{aligned}$$

### Vergleichende Anweisungen

Vergleichende Anweisungen müssen etwas anders dargestellt werden, weil hier der logische Wert des Vergleichs als Resultat zurückgegeben werden soll. Für die bool'schen Werte verwenden wir nullstellige Konstruktoren, einen mit Konstruktornummer 1 für Falsch und einen mit Konstruktornummer 2 für Wahr.

$$\begin{aligned}\iota : &\langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{Num}\ x; a_2 = \texttt{Num}\ y], *, \ldots, * \rangle\\ &\vdash \langle *, *, a : S, *, H[a = b(x \odot y)], *, \ldots, * \rangle,\\ &\forall(\iota, \odot) \in \{(\texttt{Lt}, <), (\texttt{Le}, \leq), (\texttt{Gt}, >), (\texttt{Ge}, \geq), (\texttt{Eq}, =), (\texttt{Ne}, \neq)\},\\ &b(x) = \begin{cases} \texttt{Constr 1 []}, & x = \text{Falsch}\\ \texttt{Constr 2 []}, & x = \text{Wahr}\end{cases}\end{aligned}$$

### Zeichenorientierte Anweisungen

An zeichenorientierten Anweisungen stehen `Ord` und `Chr` zur Verfügung.

$$\begin{aligned}\texttt{Ord} : &\langle *, *, a : S, *, H[a = \texttt{Constr}\ t\ \texttt{[]}], *, \ldots, * \rangle\\ \vdash\ &\langle *, *, a' : S, *, H[a' = \texttt{Num}\ t], *, \ldots, * \rangle\end{aligned}$$

Analog gilt für `Chr`:

$$\begin{aligned}\texttt{Chr} : &\langle *, *, a : S, *, H[a = \texttt{Num}\ n], *, \ldots, * \rangle\\ \vdash\ &\langle *, *, a' : S, *, H[a' = \texttt{Constr}\ n\ \texttt{[]}], *, \ldots, * \rangle\end{aligned}$$

## 4.6.6 Verzweigende Anweisungen

Die einzige vergleichende Anweisung der $G^{\#}$-Maschine ist die `Casejump` Anweisung, mit deren Hilfe auch die `if` Funktion des Sprachkerns umgesetzt werden kann:

```
if c t f  =  case c of
                  <2> -> t;
                  <1> -> f;
```

In der `Casejump` Anweisung findet Pattern Matching statt. Wenn der Wert $\perp$ auf dem Stapel liegt, dann ist das Ergebnis des Matching $\perp$, also kann in diesem Fall der Zustand unverändert bleiben.

$$\texttt{Casejump}\ cs : \langle *, *, a : S, *, H[a = \texttt{Bot}], *, \ldots, * \rangle \ \vdash \ \langle *, \ldots, * \rangle$$

Steht $\top$ an der Stapelspitze, müssen wir alle Alternativen als zutreffend ansehen, also jede Anweisungsfolge ausführen und die jeweils gemachten Änderungen am Heap übernehmen.

$$\begin{aligned} &\texttt{Casejump}\ cs : \overbrace{\langle *, C, a : S, *, H[a = \texttt{Top}], *, \ldots, * \rangle}^{s^0} \\ &\vdash \ s^n, \text{ wobei} \\ &\quad (\forall 1 \leq i \leq r) : s^{i-1} = \langle *, C, *, \ldots, * \rangle \\ &\quad \Rightarrow s^i = \mathit{eval}\ \langle *, c_i +\!\!+ [\mathit{Join}] +\!\!+ C, *, \ldots, * \rangle \end{aligned}$$

Für Vereinigungen wird es noch etwas aufwendiger. Hier muß für jede Komponente der Vereinigung die `Casejump` Anweisung ausgeführt werden.

$$\begin{aligned} &\texttt{Casejump}\ cs : \overbrace{\langle *, C, a : S, *, H[a = \texttt{<}(u_1, O_1), \ldots (u_r, O_r)\texttt{>}], *, \ldots, * \rangle}^{s^0} \\ &\vdash \ v(\langle *, *, S, *, H^r, *, \ldots, * \rangle, (u'_1, O'_1), \ldots, (u'_r, O'_r)), \text{ wobei} \\ &\quad (\forall 1 \leq i \leq r) : s^{i-1} = \langle *, C, a : S, *, H^{i-1}, *, \ldots, * \rangle \\ &\quad \Rightarrow s^i = \mathit{eval}\ \langle *, \texttt{Casejump}\ \ cs : \texttt{Join} : C, u_i : S, *, H^{i-1}, *, *, *, O_i, * \rangle \\ &\qquad = \langle *, C, u'_i : S, *, H^i, *, *, *, O'_i, * \rangle \end{aligned}$$

Wenn ein Verweis auf dem Stapel liegt, dann muß die $G^\#$-Maschine diesem folgen.

$$\begin{aligned} &\texttt{Casejump}\ cs : \langle *, C, a : S, *, H[a = \texttt{Ind}\ a'], *, \ldots, * \rangle \\ &\vdash \ \langle *, \texttt{Casejump}\ cs : C, a' : S, *, \ldots, * \rangle \end{aligned}$$

Befindet sich ein Konstruktor auf dem Stapel, kann anhand seiner Nummer die passende Anweisungsfolge ausgewählt werden, und diese wird dann vor den anderen noch auszuführenden Anweisungen ausgeführt.

$$\begin{aligned} &\texttt{Casejump}\ [t_1 := c_1, \ldots, t_n := c_n] : \langle *, C, a : S, *, H[a = \texttt{Constr}\ t\ cs], *, \ldots, * \rangle \\ &\vdash \ \langle *, c' : C, a : S, *, \ldots, * \rangle, \text{ wobei} \\ &\quad c' = \begin{cases} c_i, & \exists 1 \leq i \leq n : t_i = t \\ [\texttt{Unwind}], & \text{sonst} \end{cases} \end{aligned}$$

### 4.6.7 `Print`

Die `Print` Anweisung wandelt den Knoten an der Stapelspitze in `Iseq`s, die dann als Zeichenfolge angezeigt werden können.

$\texttt{Print} : \langle \omega, *, a : S, *, H[a = n], *, \ldots, * \rangle \ \vdash \ \langle \omega +\!\!+\, s(n), *, S, *, \ldots, * \rangle$, wobei $+\!\!+$ die Konkatenation von `Iseq`s ist und $s$ die Funktion, die Knoten anzeigt.

### 4.6.8 `Alloc`

Mit `Alloc` $n$ können $n$ Knoten auf dem Heap angelegt werden und ihre $n$ Adressen werden auf den Stapel gelegt. Verwendet wird diese Anweisung in Übersetzungen von `letrec` Konstrukten, um die Knoten für die lokalen Definitionen anzulegen. Die Knoten werden mit `Num 1` initialisiert.

$$\begin{array}{l} \texttt{Alloc}\ n : \langle *, *, S, *, H, *, \ldots, * \rangle \\ \quad \vdash \ \langle *, *, a_1 : \ldots : a_n : S, *, H \left[ \left. \begin{array}{c} a_1 \\ \vdots \\ a_n \end{array} \right\} = \texttt{Num 1} \right], *, \ldots, * \rangle \end{array}$$

### 4.6.9 `AbsEq`

`AbsEq` kann im Gegensatz zu `Eq` (siehe oben) für den Vergleich abstrakter Werte verwendet werden. `Eq` soll die Funktion `==` implementieren, muß also $\top$ ergeben, wenn eines der Argumente $\top$ ist, unabhängig vom anderen Argument.

`AbsEq` folgt Verweisen:

$$\begin{array}{l} \texttt{AbsEq} : \langle *, C, a_1 : a_2 : S, *, H[a_1 = \texttt{Ind}\ \ a_1'], *, \ldots, * \rangle \\ \quad \vdash \ \langle *, \texttt{AbsEq} : C, a_1' : a_2 : S, *, \ldots, * \rangle \end{array}$$

und entsprechend für $a_2$. Ansonsten gibt `AbsEq` den Wert des Vergleichs der Knoten zurück.

$$\begin{array}{l} \texttt{AbsEq} : \langle *, C, a_1 : a_2 : S, *, H[a_1 = \nu_1; a_2 = \nu_2], *, \ldots, * \rangle \\ \quad \vdash \ \langle *, \texttt{AbsEq} : C, a' : S, *, H[a' = b(\nu_1 == \nu_2)], *, \ldots, * \rangle, \text{ wobei} \\ \qquad b(x) = \begin{cases} \texttt{Constr}\ 1, & x = \text{ Falsch} \\ \texttt{Constr}\ 2, & x = \text{ Wahr} \end{cases} \end{array}$$

### 4.6.10 `Join`

Die `Join` Anweisung nimmt keine Änderung des Zustandes vor. Sie sorgt lediglich dafür, daß der folgende Zustand als Endzustand betrachtet wird.

$$\texttt{Join} : \langle *, \ldots, * \rangle \ \vdash \ \langle *, \ldots, * \rangle$$

### 4.6.11 `Update`

Die `Update` $n$ Anweisung und die `Unwind` Anweisung sind die mächtigsten Anweisungen in der $G^{\#}$-Maschine. Bei `Update` wird die Markierung der Wurzel entfernt, die überschrieben wird und wenn die Wurzel zu den offenen Redexen gehört, wird sie und alle nach der Wurzel eröffneten Redexe aus den offenen entfernt. Ebenso werden alle Verweise auf die Wurzel aus den offenen Redexen entfernt. Es wird geprüft, ob $\perp$-Einführung stattfinden muß und wenn die Wurzel markiert war, wird dafür gesorgt, daß die Auswertung unterbrochen wird.

$$\begin{array}{l}
\texttt{Update}\, n : \langle *, C, a : a_0 : \ldots : a_n : S, *, H\left[\left.\begin{array}{c} h_1 \\ \vdots \\ h_s \end{array}\right\} \mathrm{ind}^* a\right], *, *, *, O, M\rangle \\
\quad \vdash \quad \langle *, C', a : a_0 : \ldots : a_n : S, *, H[a_n = \nu], *, *, *, O', M \setminus a_n\rangle, \text{ wobei}
\end{array}$$

$$h \;\mathrm{ind}^i a \iff (\exists b_1, \ldots, b_i) : h = b_1 \wedge H\left[\begin{array}{ccc} b_1 & = & \texttt{Ind}\ b_2 \\ & \vdots & \\ b_i & = & \texttt{Ind}\ a \end{array}\right]$$

$$h \;\mathrm{ind}^* a \iff (\exists i) : h \;\mathrm{ind}^i a$$

$$\nu = \begin{cases} \texttt{Bot}, & \text{wenn in jedem Reduktionspfad von } a \text{ ein Verweis auf} \\ & a_n \text{ enthalten ist.} \\ \texttt{Ind}\ a, & \text{sonst} \end{cases}$$

$$C' = \begin{cases} \texttt{Join} : C, & \text{wenn } a_n \in M \\ C, & \text{sonst} \end{cases}$$

$$O' = \begin{cases} [o_{i+1}, \ldots, o_m] \cap [h_1, \ldots, h_s], & \text{wenn } O = [o_1, \ldots, o_m] \wedge \\ & (\exists 1 \le i \le m) : o_j = a \\ O, & \text{sonst} \end{cases}$$

### 4.6.12 `Unwind`

Für `Unwind` unterscheiden wir zunächst, ob der Dump leer ist oder nicht. Ist der Dump leer und es befindet sich ein Ausdruck in WHNF auf dem Stapel, muß keine Änderung erfolgen.

$$\texttt{Unwind} : \langle *, *, a : S, [], H\left[a = \begin{cases} \texttt{Top} \\ \texttt{Bot} \\ \texttt{Num}\ n \\ \texttt{Constr}\ t\ as \end{cases}\right], *, \ldots, *\rangle \quad \vdash \quad \langle *, \ldots, *\rangle$$

Ist der Dump nicht leer, wird Code und Stapelinhalt vom Dump in den Zustand übernommen.

$$\begin{array}{l}\texttt{Unwind}:\langle *,*,a:S,(C',S'):D,H\left[a=\left\{\begin{array}{l}\texttt{Top}\\\texttt{Bot}\\\texttt{Num}\ n\\\texttt{Constr}\ t\ as\end{array}\right.\right],*,\ldots,*\rangle\\
\quad\vdash\ \langle *,C' +\!\!+\, C,a:Z,D,*,\ldots,*\rangle,\text{ wobei}\\
\qquad Z=\left\{\begin{array}{ll}S', & \text{wenn } H\left[a=\left\{\begin{array}{l}\texttt{Top}\\\texttt{Bot}\end{array}\right.\right]\\ S +\!\!+\, S', & \text{sonst}\end{array}\right.\end{array}$$

Anwendungsknoten werden durch einen Verweis auf einen offenen Redex ersetzt, wenn die Anwendung kleiner ist als der bereits offene Redex. In diesem Fall wird auch, wenn der Anwendungsknoten markiert bzw. nicht markiert war, der Code gelöscht bzw. durch ein einzelnes `Unwind` ersetzt. Ist die Anwendung nicht kleiner als einer der offenen Redexe, wird ihr erster Verweis auf den Stapel gelegt, vor die noch auszuführenden Anweisungen ein `Unwind` gesetzt und wenn es sich bei der Anwendung um einen Redex handelt, wird er den offenen Redexen hinzugefügt.

$$\begin{array}{l}\texttt{Unwind}:\langle *,C,a:S,*,H[a=\texttt{Ap}\ a_1\ a_2],*,*,*,O,M\rangle\\
\quad\vdash\ \langle *,C',S',*,H',*,*,*,O',*\rangle,\\
\qquad S'=\left\{\begin{array}{ll}a:S, & \text{wenn } (\exists o\in O):a\leq^{\#} o\\ a_1:a:S, & \text{sonst}\end{array}\right.\\
\qquad H'=\left\{\begin{array}{ll}H[a=\texttt{Ind}\ o], & \text{wenn } (\exists o\in O):a\leq^{\#} o\\ H, & \text{sonst}\end{array}\right.\\
\qquad C'=\left\{\begin{array}{ll}[], & \text{wenn } (\exists o\in O):a\leq^{\#} o\wedge a\in M\\ [\texttt{Unwind}], & \text{wenn } (\exists o\in O):a\leq^{\#} o\wedge a\notin M\\ \texttt{Unwind}:C, & \text{sonst}\end{array}\right.\\
\qquad O'=\left\{\begin{array}{ll}a:O, & \text{wenn } (\forall o\in O):a\not\leq^{\#} o\wedge \texttt{Ap}\ a_1\ a_2 \text{ ist Redex}\\ O, & \text{sonst}\end{array}\right.\end{array}$$

Bei Verweisen müssen wir nur beachten, ob die Adresse auf die verwiesen wird von einem der offenen Redexe ist oder nicht.

$$\begin{array}{l}\texttt{Unwind}:\langle *,C,a:S,*,H[a=\texttt{Ind}\ a_1],*,*,*,O,*\rangle\\
\quad\vdash\ \langle *,\texttt{Unwind}:C,a':S,*,H',*,\ldots,*\rangle,\text{ wobei}\\
\qquad\begin{array}{lcl}H' &=& H[a'=\texttt{Bot}],\text{ wenn } a\in O\\ \left.\begin{array}{lcl}a' &=& a_1,\\ H' &=& H,\end{array}\right\} & & \text{sonst}\end{array}\end{array}$$

Bei Vereinigungen müssen wir prüfen, ob eine der Komponenten die Resourcen bereits erschöpft hat oder nicht und ob es sich um einen Redex handelt oder nicht.

$$\begin{array}{l}\texttt{Unwind}:\langle *,C,a:S,*,H[a=\texttt{<}(u_1,O_1),\ldots,(u_n,O_n)\texttt{>}],*,*,*,O,*\rangle\\
\quad\vdash\ \langle *,[],*,\ldots,*\rangle,\text{ wenn}\\
\qquad(\exists 1\leq i\leq n):(u_i,O_i)\text{ erschöpft die Resourcen.}\end{array}$$

Ist das nicht der Fall, werden also die Resourcen noch nicht erschöpft.

$$\begin{array}{ll} \multicolumn{2}{l}{\texttt{Unwind} : \langle *, C, a : S, *, H[a = x@\texttt{<}(u_1, O_1), \ldots, (u_n, O_n)\texttt{>}], *, *, *, O, * \rangle} \\ \vdash & v(\langle *, [\texttt{Unwind}], S, *, H^n, *, \ldots, * \rangle, (u_1, O_1'), \ldots, (u_n, O_n')), \\ & \text{wenn } x \text{ ein Redex ist und} \\ & (1 \leq i \leq n) : \mathit{eval}\ \langle *, \texttt{Unwind} : C, u_i : S, *, H^{i-1}, *, *, O_i, u_i : M \rangle \\ & \qquad\qquad = \langle *, *, *, *, H^i, *, *, O_i', * \rangle \end{array}$$

Hier können im resultierenden Zustand die gleichen Heapadressen $u_i$ verwendet werden, wie im Ausgangszustand, denn das Auswerten mit der markierten Wurzel läuft genau so lange bis die Wurzel $u_i$ updated wird.

Trifft keiner der beiden vorhergehenden Fälle zu wird der Zustand wie für `Constr` bzw. `Num` Knoten verändert.

Befindet sich schließlich ein `Global` Knoten an der Stapelspitze, unterscheiden wir, ob ausreichend Argumente auf dem Stapel liegen, um den Superkombinator anzuwenden oder nicht.

$$\begin{array}{ll} \multicolumn{2}{l}{\texttt{Unwind} : \langle *, C, a : a_1 : \ldots : a_n : S, *, H \left[ \begin{array}{lcl} a & = & \texttt{Global}\ n\ C' \\ a_n & = & \texttt{Ap}\ a_{n-1}\ a_n' \\ & \vdots & \\ a_1 & = & \texttt{Ap}\ a\ a_1' \end{array} \right], *, \ldots, * \rangle} \\ \vdash & \langle *, C' +\!\!\!+\, C, a_1' : \ldots : a_n' : S, *, *, \ldots, * \rangle \end{array}$$

Sind jedoch weniger als $n$ Einträge auf dem Stapel, wird der gleiche Zustandsübergang wie für `Constr` oder `Num` Knoten angewandt.

# Kapitel 5

# Andere Methoden

Für Striktheits-Analyse können auch andere Methoden zum Einsatz kommen und wir wollen zwei davon hier besprechen: das Propagieren strikter Kontexte und die Striktheits-Analyse mittels abstrakter Interpretation. Im Anschluß daran werden wir anhand der `standard.prelude` des Gofer-Interpreters die Methoden der abstrakten Interpretation und der abstrakten Reduktion miteinander Vergleichen.

## 5.1 Strikte Kontexte

Das Propagieren strikter Kontexte ist in [PJL91] sehr anschaulich dargelegt. Bei dieser Methode werden zwei Kontexte unterschieden in denen ein Ausdruck auftreten kann. Nämlich:

- der *strikte*, in dem der Wert des Ausdrucks in WHNF benötigt wird und
- der *nicht-strikte*, in dem nicht bekannt ist, ob der Wert des Ausdrucks in WHNF benötigt wird.

Der Kontext eines Ausdrucks ergibt sich dann aus einigen einfachen Regeln.

- Der Ausdruck auf der rechten Seite eines Superkombinators ist in striktem Kontext.
- Tritt $e_0 \odot e_1$ in striktem Kontext auf, wobei $\odot$ ein arithmetischer oder vergleichender Operator ist, dann sind $e_0$ und $e_1$ in striktem Kontext.
- Tritt $f\ e$ in striktem Kontext auf, wobei $f \in \{\texttt{negate}, \texttt{ord}, \texttt{chr}\}$, so ist $e$ ebenfalls in striktem Kontext.
- Tritt `case` $e$ `of` $alt_1$`;` $\ldots alt_n$`;` in striktem Kontext auf, so befinden sich auch $e, alt_1, \ldots alt_n$ in striktem Kontext.
- Tritt schließlich `let(rec)` $\Delta$ `in` $e$ in striktem Kontext auf, so befindet sich $e$ in striktem Kontext.

Diese einfachen Regeln ermöglichen bereits die Zahl der verwendeten Heap-Knoten und die Zahl der G-Maschinen Anweisungen, die verwendet werden deutlich einzuschränken.

## 5.2 Abstrakte Interpretation

Eine weitere Methode zur Striktheits-Analyse verwendet abstrakte Interpretation. Bei der abstrakten Interpretation wird der Sprache eine denotationale Semantik gegeben, die sich von der Standard-Semantik nur durch die Interpretation der Konstanten und eingebauten Funktionen unterscheidet. Es wird eine zwei-elementige Domäne verwendet, in der $\perp^{\#}$ die Abstraktion von $\perp$ und $\top^{\#}$ die Abstraktion von allen Werten außer $\perp$ bezeichnet. Die abstrakte Interpretation einer (nicht-rekursiven) Funktion kann dann durch einige naheliegende Regeln erzeugt werden.

Handelt es sich bei einem Ausdruck um eine Anwendung, so ist seine abstrakte Interpretation die Anwendung der abstrakt interpretierten Funktion auf das abstrakt interpretierte Argument. Hierfür schreibt man (siehe [PJ87]):

$$\mathsf{Eval}^{\#}[\![e_0\ e_1]\!] = \mathsf{Eval}^{\#}[\![e_0]\!]\ \mathsf{Eval}^{\#}[\![e_1]\!]$$

Für formale Parameter $v$ von Funktionen gilt

$$\mathsf{Eval}^{\#}[\![v]\!] = v$$

und für Konstanten $k$

$$\mathsf{Eval}^{\#}[\![k]\!] = \top^{\#}.$$

Wir bezeichnen mit $\otimes$ den zweistelligen Operator auf der abstrakten Domäne, der $\top^{\#}$ ergibt, genau dann wenn beide Operanden $\top^{\#}$ sind und mit $\oplus$ den ebenfalls zweistelligen Operator, der $\perp^{\#}$ ergibt, genau dann wenn beide Operanden $\perp^{\#}$ sind. Um Klammern zu sparen geben wir $\otimes$ Vorrang vor $\oplus$.

Die abstrakten Interpretationen der primitiven Funktionen geben wir vor:

$$\texttt{==}^{\#}\ x\ y = x \otimes y$$

und entsprechendes für die anderen vergleichenden und arithmetischen Operatoren.

Das `case`-Konstrukt hat die abstrakte Interpretation

$$\mathsf{Eval}^{\#}[\![\texttt{case}\ e\ \texttt{of}\ t_1\ \texttt{->}\ e_1\texttt{;}\ \ldots t_n\ \texttt{->}\ e_n\texttt{;}]\!] = e \otimes \bigoplus_{1 \leq i \leq n} e_i$$

Als ein Beispiel betrachten wir die Funktion

```
f p q r = if (p == 0) (q + r) (q + p)
```

Die abstrakte Interpretation ist (nach der Regel für Anwendungen)

$$\texttt{f}^{\#}\ p\ q\ r = \texttt{if}^{\#}\ (p\ \texttt{==}^{\#}\ \top^{\#})\ (q\ \texttt{+}^{\#}\ r)\ (q\ \texttt{+}^{\#}\ p)$$

Ist `if` definiert als

```
if c t f  =  case c of
                  <1> -> f;
                  <2> -> t;
```

ergibt sich für $\texttt{if}^\#$

$$\texttt{if}^\#\ c\ t\ f = c \otimes (t \oplus f)$$

Also für $\texttt{f}^\#$

$$\texttt{f}^\#\ p\ q\ r = (p\ \texttt{==}^\#\ \top^\#) \otimes ((q\ \texttt{+}^\#\ r) \oplus (q\ \texttt{+}^\#\ p))$$

und aus den Regeln für die primitiven Funktionen erhalten wir

$$\begin{array}{rcl} \texttt{f}^\#\ p\ q\ r & = & (p \oplus \top^\#) \otimes (q \otimes r \oplus q \otimes p) \\ & = & p \otimes (q \otimes (p \oplus r)) \end{array}$$

Um daraus Striktheits-Information zu erhalten, müssen wir lediglich $\top^\#$ bzw. $\perp^\#$ an den entsprechenden Stellen einsetzen. Wir erhalten so:

$$\begin{array}{rcccl} \texttt{f}^\#\ \perp^\#\ \top^\#\ \top^\# & = & \perp^\# \otimes (\top^\# \otimes (\perp^\# \oplus \top^\#)) & = & \perp^\# \\ \texttt{f}^\#\ \top^\#\ \perp^\#\ \top^\# & = & \top^\# \otimes (\perp^\# \otimes (\top^\# \oplus \top^\#)) & = & \perp^\# \\ \texttt{f}^\#\ \top^\#\ \top^\#\ \perp^\# & = & \top^\# \otimes (\top^\# \otimes (\top^\# \oplus \perp^\#)) & = & \top^\# \end{array}$$

`f` ist also strikt in seinen ersten beiden Argumenten. Das sieht zunächst vielversprechend aus, aber wir haben bisher noch keine rekursiven Funktionen verwendet.

Um rekursive Funktionen zu untersuchen, ist es notwendig eine aufsteigende Folge von Approximationen zu $\texttt{f}^\#$ zu erzeugen.

$$\begin{array}{lcl} \texttt{f}^\#\ a_1\ \ldots\ a_n & = & \ldots\ \texttt{f}^\#\ \ldots \\ \multicolumn{3}{l}{\text{wird approximiert durch}} \\ \texttt{f}_1^\#\ a_1\ \ldots\ a_n & = & \perp^\# \\ \texttt{f}_2^\#\ a_1\ \ldots\ a_n & = & \ldots\ \texttt{f}_1^\#\ \ldots \\ & \vdots & \end{array}$$

Da es nur endlich viele $n$-stellige Funktionen über der abstrakten Domäne gibt, erreicht diese Folge in endlich vielen Schritten einen Fixpunkt, den wir nur feststellen können, indem wir aufeinanderfolgende Approximationen auf Gleichheit überprüfen, was im schlechtesten Fall Kosten exponentiell in der Zahl der Argumente verursacht.

Für Funktionen höherer Ordnung wird die Berechnung der abstrakten Interpretation noch teurer, denn im Test auf Gleichheit müssen für funktionswertige Argumente alle in den Approximationen auftretenden Belegungen geprüft werden und nicht nur die beiden Werte der abstrakten Domäne.

## 5.3 Vergleich

Eine Frage, die wir bisher nicht beantwortet haben lautet: Findet unsere Implementation Striktheits-Information, die Implementationen anderer Methoden nicht finden? Wir wollen sie hier beantworten.

Als Repräsentant der anderen Methoden stand uns der Chalmers Haskell B. Compiler in der Version 0.999.5 zur Verfügung, der mit einer nicht-dokumentierten Option ([Aug94]) dazu gebracht werden kann, die ermittelte Striktheits-Information auszugeben. Im Haskell B. Compiler wird ausschließlich abstrakte Interpretation verwendet ([Aug94]). Für den Vergleich wirde der Haskell B. Compiler mit dem Kommando

```
hbc -no-pedantic -fstrictinfo -fprelude -c standard.prelude.hs
```

aufgerufen. Dabei fand er 93 strikte Argumente. Unsere Implementation fand darüber hinaus 21 weitere.

Einige davon sind Argumente von

```
gcd x y   =  gcd´ (abs x) (abs y)
gcd´ x y  =  if (y == 0)
                x
                (gcd´ y (x `rem` y))
```

und

```
until p f x  =  if (p x)
                   x
                   (until p f (f x))
```

Aber auch für Funktionen, die sich nicht in der `standard.prelude` befinden, wie folgende Variante von `length`, fand der hbc nicht alle strikten Argumente.

```
length x n  =  case x of
                  <1> -> n;
                  <2> a b -> length b (n + 1);
```

`gcd` und `gcd´` sind strikt in beiden Argumenten, `until` ist strikt im ersten und `length` ist ebenfalls strikt in beiden Argumenten. Der hbc stellte für keines der Argumente dieser Funktionen Strktheit fest, unsere Implementation ermittelte in allen diesen Fällen die vollständige Striktheits-Information.

# Kapitel 6

# Zusammenfassung und Ausblick

Die Implementation der Striktheits-Analyse, die im Zuge dieser Arbeit vorgenommen wurde, stellt eine effiziente Approximation der abstrakten Reduktion mit Pfadanalyse dar. Durch die $G^{\#}$-Maschine, ein neues, auf der G-Maschine basierendes Maschinenmodell, wurde die verwendete Methode systematisch dargelegt. Die große Ähnlichkeit mit der G-Maschine, die in unserer Implementation beibehalten werden konnte, zeigt, wie natürlich die verwendete Methode der Reduktion in funktionalen Programmiersprachen entspricht. Obwohl die Umsetzung mehr Wert auf Nachvollziehbarkeit, als auf Effizienz legt, zeigt sie, daß die Methode der abstrakten Reduktion mit Pfadanalyse auch in einer funktionalen Implementierung durchaus alltagstauglich ist und Striktheits-Information findet, die Umsetzungen anderer Methoden nicht finden. Es bestehen Möglichkeiten zur Optimierung u. a. von Programmteilen, die für jede simulierte $G^{\#}$-Maschinen-Anweisung ausgeführt werden. Bei vorsichtiger Einschätzung erscheint eine Halbierung der Laufzeit mit vertretbarem Aufwand erreichbar.

## 6.1 Resultate

Die Implementation wurde auf einem z. Zt. handelsüblichen PC mit einem Intel 486 DX2/66 Prozessor und 16 MB Hauptspeicher unter Linux Version 1.0.7 und Version 1.1.24 mit dem Chalmers Haskell B. Compiler Version 0.999.5 übersetzt und ausgeführt. Zum Test wurde eine Datei verwendet, die

- die `standard.prelude` des Gofer Interpreters mit Ausnahme der Fließkomma-Funktionen und der Typklassen,
- den Compiler vom Sprachkern zu $G^{\#}$-Code,
- den Lexer für den Sprachkern und
- einige Hilfsfunktionen

umfaßt. Bei diesem Test wurden in ca. 32 Sekunden die 287 Funktionen analysiert. Die Ausgabe der Analyse durch das Kommando

```
time Analyser -gc-gen ⟨name⟩.core 2>&1 >⟨name⟩.strict
```

findet sich im Anhang B.

## 6.2 Ausblick

### 6.2.1 Kontexte

Striktheits-Analyse mit der hier verwendeten Methode kann einen großen Beitrag zur Erzeugung von effizientem Code leisten.

Wie können wir den noch vergrößern?

In einigen Fällen ergibt sich kein besonderer Vorteil aus der Kenntnis der Striktheits-Information. Ein einfaches Beispiel hierfür ist `sum`:

```
sum xs  =  case xs of
              <1> -> 0;
              <2> a b -> a + (sum b);
```

Wird das Ergebnis eines Aufrufs `sum` $x$ in WHNF benötigt, wird natürlich auch $x$ in WHNF benötigt. Wir können aber noch viel mehr sagen. Es wird die gesamte Liste betrachtet und jedes Element wird in WHNF benötigt. Wir sagen, daß Funktionen, die $x$ darstellen, in einem *Kontext* auszuwerten sind, der in diesem Fall die Auswertung des Rückgrates der Liste und die Auswertung ihrer Elemente verlangt. Dies kann man für andere strukturierte Datentypen verallgemeinern.

Das allgemeine Striktheitsproblem sucht also nach dem Kontext $e$, in dem ein Argument ausgewertet werden kann, wenn die Anwendung in einem Kontext $f$ auszuwerten ist. Die hier verwendete Methode ist zur Simulation solcher Kontexte in der Lage. Dies kann durch zusätzliche Funktionen (Kontextfunktionen) und bestimmte abstrakte Werte geschehen. Im Beispiel:

```
e xs  =  case xs of
            <1> -> Cons{1,0};
            <2> a b -> k (e b) a;


k a b  =  case a of
             _ ->   case b of
                       _ -> a;
```

untersuchen wir

```
letrec topmem  =  <Cons{1,0}, Cons{2,2} ⊤ topmem>;
       botmem  =  <Cons{2,2} ⊥ topmem, Cons{2,2} ⊤ botmem>;
             in e (append botmem topmem);
```

Dabei stellt `botmem` die Listen dar, die entweder unendlich lang sind oder ein $\perp$ enthalten. Wird für diesen Ausdruck $\perp$ (abstrakt) reduziert, bedeutet das, daß für eine Anwendung von `append` im von `e` dargestellten Kontext das erste Argument im von `botmem` dargestellten Kontext ausgewertet werden kann. Beide stellen im Beispiel den gleichen Kontext dar. Es ist möglich, für strukturierte Datentypen die Kontextfunktion und den darstellenden abstrakten Wert automatisch abzuleiten.

Es ist jedoch für solche allgemeine Striktheits-Information nicht immer offensichtlich, wie sie zur Beschleunigung des erzeugten Programmes eingesetzt werden kann. Ist uns zum Beispiel bekannt, daß `sum` die gesamte Argumentliste betrachten muß, so können wir diese vollständig im Speicher abrollen. Alle Vorteile werden jedoch schnell zunichte, wenn wir deshalb gezwungen sind, ein Vielfaches der gewonnenen Zeit mit Garbage Collection zu verbringen.

Eine wichtige offene Frage ist also: Wie kann allgemeine Striktheits-Information zur Beschleunigung des erzeugten Programms verwendet werden.

### 6.2.2 Modularisieren

Im allgemeinen liegt auch bei funktionalen Sprachen nicht der ganze Programmtext in einer Datei vor, sondern es werden Module verwendet, um ihn zu strukturieren. Um die Striktheits-Analyse zu erweitern, damit sie mit Modulen arbeitet, bedarf es zunächst keiner allzu großen Änderungen. Es kann die Ausgabe der Analyse eines Moduls wieder eingelesen werden und für die Funktionen darin $G^{\#}$-Code angelegt werden, der sich sehr ähnlich dem `analyserCode` verhält. Im Unterschied zu diesem soll dann aber nicht der $G^{\#}$-Code der ursprünglichen Funktion angesprungen werden, sondern es soll, wenn an keiner strikten Stelle $\perp$ auftritt, $\top$ zurückgegeben werden. Dadurch geht natürlich Information verloren, die für Funktionen im gleichen Modul vorhanden ist, und dies verschlechtert das Resultat der Striktheits-Analyse. Ein Beispiel für die Verschlechterung stammt aus der `standard.prelude`:

```
iterate f x     =  Cons{2,2} x (iterate f (f x));
takeUntil p ys  =  case ys of
                      <1> -> Cons{1,0};
                      <2> a b -> if (p a)
                                    (Cons{2,2} a Cons{1,0})
                                    (Cons{2,2} a (takeUntil p b));
until´ p f x    =  takeUntil p (iterate f x);
```

Betrachten wir das erste Argument von `until´`, wenn `takeuntil` bereits analysiert ist:

$$\texttt{until´}\ \perp\ \top\ \top\ \ \to_\alpha\ \ \texttt{takeUntil}\ \perp\ \texttt{(iterate}\ \top\ \top\texttt{)}$$

Unser Programm findet für `takeUntil` Striktheit im zweiten Argument. Approximieren wir also, weil keine strikte Position von `takeUntil` mit $\perp$ besetzt ist, mit $\top$,

$$\texttt{takeUntil}\ \perp\ \texttt{(iterate}\ \top\ \top\texttt{)}\ \ \to_\alpha\ \ \top$$

finden wir keine Striktheit für das erste Argument von `until´`. Anders, wenn wir die Ersetzungsregel für `takeUntil` verwenden:

```
takeUntil ⊥ (iterate ⊤ ⊤)
  →α  if (⊥ ⊤)
        (Cons{2,2} ⊤ Cons{1,0})
        (Cons{2,2} ⊤ (takeUntil ⊥(iterate ⊤ (⊤ ⊤))))
  →α  ⊥, weil (⊥ ⊤) an strikter Position
```

Hier wird also Striktheit im ersten Argument von `until´` gefunden.

Woran liegt dieser Unterschied?

`takeUntil` ist nicht strikt im ersten Argument, aber

```
\x -> takeUntil x (iterate ⊤ ⊤)
```

ist strikt. Im ersten Fall geht uns diese Information durch die Approximation verloren, aber da wir im zweiten Fall abstrakt weiterreduzieren, finden wir schließlich Striktheit.

Es bleibt die Frage offen, wie die Striktheits-Analyse in geeigneter Weise für Module erweitert werden kann.

### 6.2.3 Variation des Maschinenmodells

Die G-Maschine ist, seit sie von Augustsson und Johnsson vorgestellt wurde, von vielen Autoren weiterentwickelt worden. Dabei standen meistens Effizienzsteigerungen im Vordergrund. Aufgrund der oben erwähnten Verwandtschaft der abstrakten Reduktion und der Reduktion drängt sich die Frage auf, inwieweit mit den neueren Maschinenmodellen, wie z. B. der ABC-Maschine, der $\langle\nu, G\rangle$-Maschine oder der STG-Maschine die abstrakte Reduktion dargestellt werden kann. Man kann erwarten, daß auch die Striktheit in diesen Modellen effizienter untersucht werden kann.

### 6.2.4 Kompilieren

Die Analyse verbringt den größten Teil ihrer Laufzeit mit der Simulation der $G^{\#}$-Maschinen-Anweisungen. Es erscheint also sinnvoll, anstatt die Anweisungen zu simulieren, Maschinencode oder C-Code zu erzeugen, der dann ausgeführt wird. Vor allem für große Programme könnte sich hier eine deutliche Beschleunigung der Analyse ergeben.

### 6.2.5 Verwendung mehrerer Verfahren

Eine weitere Möglichkeit, eine deutlliche Beschleunigung zu erzielen, ist die Verwendung mehrerer verschiedener Verfahren zur Striktheits-Analyse. Im Besonderen kann der Compiler bei der Übersetzung nach $G^{\#}$-Code in bestimmten Fällen Striktheit sehr leicht feststellen. Es ist offensichtlich vorteilhaft, diese Striktheits-Information zu nutzen, um Striktheits-Analyse mittels abstrakter Reduktion nur noch für Argumente vorzunehmen, bei denen nicht schon mit einfachen Mitteln Striktheit gefunden werden konnte.

Gibt es auch Fälle, in denen mit einfachen Mitteln festgestellt werden kann, daß eine Funktion in einem Argument nicht strikt sein kann? Ja, ein Beispiel für solche Funktionen ist:

```
k a b = a
```

Hier kann man leicht feststellen, daß `k` in `b` nicht strikt[1] sein kann, denn `b` wird auf der rechten Seite nicht verwendet. Ein anderer Fall wären die Funktionen, deren rechte Seite bereits offensichtlich in WHNF sind. Z. B. wenn die rechte Seite ein Konstruktor mit seinen Argumenten ist oder eine Funktionsanwendung, mit weniger Argumenten als der Stelligkeit der Funktion. Es bleibt offen, in welchen anderen Fällen wir mit einfacheren Methoden Striktheits-Information finden können.

[1] `k` könnte gemäß unserer Definition strikt in `b` sein, wenn `k` immer nicht-terminierend wäre, aber eine Beschleunigung könnten wir durch die Striktheits-Information nicht erzielen.

# Anhang A

# Programm

Dieser Anhang enthält einige wesentliche Teile des Programms zur Striktheits-Analyse. Der vollständige Programmtext ist auf Wunsch vom Autor, unter der auf der Titelseite erwähnten Adresse, erhältlich. Die Module, die nicht abgedruckt wurden, beziehen sich auf den Lexer, den Compiler, die Hilfsfunktionen und die Funktionen zur Ausgabe der Ergebnisse.

Von der `main`-Funktion werden alle Superkombinatoren einer Datei analysiert. In der Gofer Version fragt sie den Anwender nach einem Namen, in der Haskell Version wird das Kommandozeilenargument verwendet. In beiden Versionen wird dann die Datei gelesen und die enthaltenen Superkombinatoren werden analysiert.

```
> module Main where
> import Impl (allNames, strictPosition,
>              callGraph, callGraphSCCs,
>              CallGraphEdge(..), analyseFunctions,
>              analyserStartStates, CallGraph(..), SCC(..))
> import Compiler
> import Evaluator
> import Show
> import IO
> import StateType
> import AbsSyntax
> import Either
> import Maybe


> main :: Dialogue
> main ~(StrList progArgs : ~(r3 :_)) =
>  [
>   GetArgs,
>    ReadFile name,
>    AppendChan stdout
>       (case r3 of
```

```
>           Str contents ->
>               let initialState =
>                       putOpenTermLimit (convert depth) (compile (parse contents))
>               in
>                   (showAnalyserResults [analyseFunctions
>                    (allNames initialState)
>                    initialState]) ++ "\n"
>           Failure _ -> "Can´t open " ++ name)]
>   where
>      (depth : name :_) = progArgs
>
> convert :: String -> Int
> convert ss = foldl multadd 0 [(ord s) - (ord ´0´) | s <- ss]
>   where multadd z x = z*10+x
```

```
> module Impl where
```

(StateType.., showShortResults, analyseFunctions, compile, allNames, callGraph, callGraphSCCs, CallGraphEdge(..), showAnalyserResults) where

```
> import QSort
> import Utils
> import AbsSyntax
> import Show(showResults, showShortResults, showAnalyserResults,
>       showHiLevel, showState)
> import Compiler(compile)
> import Evaluator(eval, neq, resourcesExhausted)
> import StateType
```

# A.1 Die $G^{\#}$-Maschine

## A.1.1 Die grobe Struktur

An der Oberfläche präsentiert sich die $G^{\#}$-Maschine wie ihr konkretes Gegenstück, nur daß hier drei verschiedenen Aufrufe zur Verfügung stehen, je nach gewünschtem Detail bei der Ausgabe.

Zur Verfügung stehen:

- Die Funktion `hrun` (steht für high-level run) zeigt nur den Ausdruck nach jedem Ersetzungsschritt an.

  ```
  > hrun :: [Char] -> [Char]
  > hrun = showHiLevel . eval . compile . parse
  ```

- Die Funktion `drun` (steht für detailed run) gibt jeden Zustand der abstrakte Maschine im Detail aus.

```
> drun :: [Char] -> [Char]
> drun = showResults . eval . compile . parse
```

- Die Funktion `crun` (das steht für compile and run) gibt nur den Endzustand aus.

```
> crun :: [Char] -> [Char]
> crun = showShortResults . eval . compile . parse
```

### A.1.2 Grundlegende Datentypen

Hier finden sich der Typ für den Zustand und einige auf ihm arbeitende Funktionen.

```
> module StateType
>   (StateType.., Instruction(..),
>    Heap(..), AssocList(..), Addr(..),
>    Iseq, Name(..)) where
> import AbsSyntax
> import InstrType (Instruction(..))
> import Utils
```

Der Anfangszustand der abstrakten Maschine wird vom Compiler erzeugt. Der Typ `GmState` stellt Zustände der Maschine dar.

```
> type GmState =
>   (GmOutput,  -- Bisherige Ausgabe
>    (GmCode,   -- Derzeitiger Anweisungsfolge
>    GmStack,   -- Aktueller Stapel
>    GmDump,    -- Ein Stapel von Stapels...
>    GmHeap,    -- Heap f"ur die Graphen
>    GmGlobals),    -- Die Adressen der globalen Symbole
>                       --   im Heap
>    (GmStats,  -- Statistik
>    GmPaths,   -- Die Pfade, die uns vom letzten
>           --   Zustand zu diesem gef"uhrt haben
>    GmOpen,    -- Ausdr"ucke, die ausgewertet werden
>           --   und noch nicht in WHNF sind
>    GmOpenTermLimit,
>           -- na ja, openTermLimit eben
>    GmMarks))  -- Marken f"ur update
```

Betrachten wir die einzelnen Komponenten des Zustands:

- Die Ausgabe-Komponente hat den Typ `GmOutput`, was seinerseits ein Synonym für Iseq ist. Iseq ist ein Datentyp, der, zusammen mit seinen Funktionen eingeführt wurde, um die Kosten wiederholter Anwendungen von `++` zu dämpfen.

```
> type GmOutput = Iseq
```

  Um auf die Komponenten des Zustandes zugreifen zu können, definieren wir entsprechende Zugriffsfunktionen für jede Komponente des Zustandes. Diese ermöglichen uns den Datentyp für die Zustände zu ändern, ohne an bestehenden Funktionen außer den Zugriffsfunktionen Änderungen vornehmen zu müssen.

```
> getOutput :: GmState -> GmOutput
> getOutput (o, i, e)
>   = o


> putOutput :: GmOutput -> GmState -> GmState
> putOutput o´ (o, i, e)
>   = (o´, i, e)
```

- Der Typ von Anweisungsfolgen ist `GmCode`, was einer Liste von `Instruction`s entspricht.

```
> type GmCode = [Instruction]
```

  Die hierzu gehörenden Zugriffsfunktionen sind:

```
> getCode :: GmState -> GmCode
> getCode (o, (i, stack, dump, heap, globals), e)
>   = i


> putCode :: GmCode -> GmState -> GmState
> putCode i´ (o, (i, stk, dmp, hp, glbls), e)
>   = (o, (i´, stk, dmp, hp, glbls), e)
```

- Auf dem Stapel, vom Typ `GmStack`, liegen Adressen von Knoten im Heap,

```
> type GmStack = [Addr]
```

  auf die zugegriffen werden kann, mittels:

```
> getStack :: GmState -> GmStack
> getStack (o, (i, stk, dmp, hp, glbls), e)
>   = stk
```

und

```
> putStack :: GmStack -> GmState -> GmState
> putStack stk´ (o, (i, stk, dmp, hp, glbls), e)
>   = (o, (i, stk´, dmp, hp, glbls), e)
```

- Der Dump ist ein Stapel von `GmDumpItem`s, die den Stapel und die Anweisungen beschreiben, bei denen eine Berechnung wieder aufgenommen werden soll.

```
> type GmDump = [GmDumpItem]
> type GmDumpItem = (GmCode, GmStack)
```

Auch hier haben wir entsprechende Zugriffsfunktionen:

```
> getDump :: GmState -> GmDump
> getDump (o, (i, stk, dmp, hp, glbls), e)
>   = dmp
```

und

```
> putDump :: GmDump -> GmState -> GmState
> putDump dmp´ (o, (i, stk, dmp, hp, glbls), e)
>   = (o, (i, stk, dmp´, hp, glbls), e)
```

- Der Heap der $G^{\#}$-Maschine wird mit Hilfe des `Heap` Datentyps, wie er in [PJL91] definiert wird dargestellt. Hier verwenden wir jedoch Paare von Knoten. Der jeweils erste Knoten stellt den ursprünglich zu reduzierenden Graphen dar, während der zweite den reduzierten Graphen darstellt.

```
> type GmHeap = Heap (Node, Node)
```

Um darauf zuzugreifen verwenden wir:

```
> getHeap :: GmState -> GmHeap
> getHeap (o, (i, stk, dmp, hp, glbls), e)
>   = hp
```

```
> putHeap :: GmHeap -> GmState -> GmState
> putHeap hp´ (o, (i, stk, dmp, hp, glbls), e)
>   = (o, (i, stk, dmp, hp´, glbls), e)
```

Die reduzierte Komponente kann mit `rLookup` und `rUpdate` bearbeitet werden,

```
> rLookup :: GmHeap -> Addr -> Node
> rLookup hp = snd . hLookup hp


> rUpdate :: GmHeap -> Addr -> Node -> GmHeap
> rUpdate hp a nd = hUpdate hp a (oLookup hp a, nd)
```

die originale Komponente mit `oLookup` und `oUpdate`

```
> oUpdate :: GmHeap -> Addr -> Node -> GmHeap
> oUpdate hp a nd = hUpdate hp a (nd, rLookup hp a)


> oLookup :: GmHeap -> Addr -> Node
> oLookup hp = fst . hLookup hp
```

und schließlich beide zusammen mit `bUpdate` und `bAlloc`.

```
> bUpdate :: GmHeap -> Addr -> Node -> GmHeap
> bUpdate hp a nd = hUpdate hp a (nd, nd)


> bAlloc :: GmHeap -> Node -> (GmHeap, Addr)
> bAlloc hp nd = hAlloc hp (nd, nd)
```

Bereits die G-Maschine verwendet folgende Knoten im Heap.

```
> data Node
>   = NNum Int                -- Zahl
>   | NAp Addr Addr           -- Funtionsanwendung
>   | NGlobal Int GmCode      -- globales Symbol
>   | NInd Addr               -- Verweis
>   | NConstr Int [Addr]      -- Konstruktor
```

Die $G^{\#}$-Maschine verwendet darüberhinaus einige besondere Knoten.

```
>   | NTop                    -- Der abstrakte Wert Top
>   | NBot                    -- Der abstrakte Wert Bot
>   | NUnion [(Addr, [Addr])]-- Vereinigung abstrakter Graphen
>   | NVisited                -- f"ur´s Absuchen des Graphen
>                             --   in inEveryReductionPath
```

Knoten, die Zahlen darstellen, haben diese Zahl als Argument, Anwendungsknoten wenden ihr erstes Argument auf ihr zweites Argument an, Knoten für globale Symbole geben mit dem ersten Argument die Stelligkeit des Symbols an und das zweite sind die auszuführenden Anweisungen, das Argument des Verweisknotens ist die Adresse des Knotens auf den er verweist, updatete Knoten verweisen mit ihrem ersten Argument auf den ursprünglichen Graphen und mit dem zweiten auf den aktuellen, das erste Argument des Konstruktorknotens ist das Tag dieses Konstruktors in seinem Typ, sein zweites Argument ist die Liste seiner Argumente, Knoten für Vereinigungen vereinigen die Graphen auf die die Zeiger in ihrem Argument zeigen, die jeweils 2. Komponente der Argumente ist die Liste der offenen Redexe.

Da an einigen Stellen solche `Node`s miteinander verglichen werden müssen, müssen wir angeben, wann zwei `Node`s gleich sind:

```
> instance Eq Node where
>   NNum a       == NNum b       = a == b
>   NAp a b      == NAp c d      = a == c && b == d
>   NGlobal a b  == NGlobal c d  = a == c && b == d
>   NInd a       == NInd b       = a == b
>   NConstr a b  == NConstr c d  = a == c && b == d
>   NTop         == NTop         = True
>   NBot         == NBot         = True
>   NUnion a     == NUnion b     = a == b
>   NVisited     == NVisited     = True
>   _            == _            = False
```

- Für die globalen Symbole verwenden wir den Typ `GmGlobals`, der eine Liste von Assoziationen von Adressen mit Namen ist.

```
> type GmGlobals = AssocList Name Addr
```

Im Zugriff haben wir diese Symbole mit:

```
> getGlobals :: GmState -> GmGlobals
> getGlobals (o, (i, stk, dmp, hp, glbls), e)
>   = glbls


> putGlobals :: GmGlobals -> GmState -> GmState
> putGlobals glbls´ (o, (i, stk, dmp, hp, glbls), e)
>   = (o, (i, stk, dmp, hp, glbls´), e)
```

- Für die Statistik verwenden wir einen abstrakten Datentyp, `GmStats`, was uns erlaubt auch hier später leicht Veränderungen vornehmen zu können.

Die Schnittstelle von `GmStats` ist:

```
> statInitial  :: GmStats
> statIncSteps :: GmStats -> GmStats
> statGetSteps :: GmStats -> Int
```

mit der folgenden Implementation:

```
> type GmStats = Int
> statInitial    = 0
> statIncSteps s = s+1
> statGetSteps s = s
```

Auf die Statistik-Komponente greifen wir mit

```
> getStats :: GmState -> GmStats
> getStats (o, i, (sts, p, op, otl, mk))
>   = sts
```

und

```
> putStats :: GmStats -> GmState -> GmState
> putStats sts´ (o, i, (sts, p, op, otl, mk))
>   = (o, i, (sts´, p, op, otl, mk))
```

zu.

- Die Pfade, die zum aktuellen Zustand geführt haben, werden mit dem Typ `GmPaths` dargestellt. Entweder waren es keine Pfade vom letzten Zustand, oder es war eine Liste von Zustandsfolgen.

```
> data GmPaths = PNil | PList [[GmState]]
```

Um solche Pfade miteinander vergleichen zu können, geben wir an, wie `GmPaths` zu einer Instanz der Klasse `Eq` wird.

```
> instance Eq GmPaths
>   where
>       PNil        == PNil     = True
>       PList a     == PList b  = a == b
>       _       == _        = False
```

Auch auf `GmPaths` können wir mit

```
> getPaths :: GmState -> GmPaths
> getPaths (o, i, (st, p, op, otl, mk)) = p
```

und

```
> putPaths :: GmPaths -> GmState -> GmState
> putPaths p´ (o, i, (st, p, op, otl, mk))
>   = (o, i, (st, p´, op, otl, mk))
```

zugreifen.

- Um schließlich die offenen Auswertungen zu notieren, verwenden wir den Typ `GmOpen`. Er ist synonym mit einer Liste von Adressen im Heap.

```
> type GmOpen = [Addr]

> getOpen :: GmState -> GmOpen
> getOpen (o, i, (st, p, op, otl, mk)) = op


> putOpen :: GmOpen -> GmState -> GmState
> putOpen op´ (o, i, (st, p, op, otl, mk))
>   = (o, i, (st, p, op´, otl, mk))


> type GmOpenTermLimit = Int

> getOpenTermLimit :: GmState -> GmOpenTermLimit
> getOpenTermLimit (o, i, (st, p, op, otl, mk)) = otl


> putOpenTermLimit :: GmOpenTermLimit -> GmState -> GmState
> putOpenTermLimit otl´ (o, i, (st, p, op, otl, mk))
>   = (o, i, (st, p, op, otl´, mk))
```

- Nun benötigen wir noch Marken, die beim Update dazu dienen festzustellen, ob wir uns in einer untergeordneten Instanz des Evaluators befinden. Dafür verwenden wir `GmMarks`, einen Stapel von Adressen.

```
> type GmMarks = [Addr]

> getMarks :: GmState -> GmMarks
> getMarks (o, i, (st, p, op, otl, mk)) = mk


> putMarks mk´ (o, i, (st, p, op, otl, mk))
>   = (o, i, (st, p, op, otl, mk´))
```

### A.1.3 Der Evaluator

Der Evaluator ist der Teil des Programms, der die Interpretation des $G^{\#}$-Codes vornimmt und so die $G^{\#}$-Maschine simuliert.

```
> module Evaluator
>   (StateType.., eval, neq, resourcesExhausted) where
> import StateType
> import Show
> import AbsSyntax
> import QSort
> import Utils
> import Compiler(rearrange)
```

Der Evaluator für die $G^{\#}$-Maschine bricht, anders als der Evaluator der G-Maschine, die Auswertung ab, wenn die Resourcenbeschränkung erreicht wird. Dies ist nötig, um Terminierung der Striktheits-Analyse zu gewährleisten. Die Auswertung wird auch abgebrochen, wenn die nächste Anweisung die `Join` Anweisung ist. Dann wird jedoch die `Join` Anweisung noch entfernt.

```
> eval :: GmState -> [GmState]
> eval state = state:restStates
>   where
>   restStates
>       | gmFinal state          = []
>       | resourcesExhausted state   = []
>       | (head (getCode state) == Join) = [putCode
>           (tail (getCode state)) state]
>       | otherwise         = eval nextState
>   nextState  = step state
```

`doAdmin` zählt die Schritte der $G^{\#}$-Maschine.

```
> doAdmin :: GmState -> GmState
> doAdmin s = putStats (statIncSteps (getStats s)) s
```

#### Der Endzustand

Ein Zustand ist Endzustand, wenn `gmFinal` wahr wird.

```
> gmFinal :: GmState -> Bool
> gmFinal s = getCode s == []
```

### Ein Schritt

Die `step` Funktion führt den Zustandsübergang anhand der nächsten Anweisung durch und initialisiert die Pfad-Komponente des Zustands.

```
> step :: GmState -> GmState
```

```
> step state = dispatch i (putPaths PNil (putCode is state))
>       where (i:is) = getCode state
```

```
\indexDTT{step}%
```

Für Zeitmessungen wird `putPaths` rausgeworfen.

step state = dispatch i (putCode is state) where (i:is) = getCode state

## A.2 Der Instruktionsvorrat

Definieren wir nun einen Datentyp für die instruktionen der $G^{\#}$-Maschine.

```
> module InstrType where
> import AbsSyntax
> import Utils
```

Viele der Instruktionen der $G^{\#}$-Maschine stammen von der G-Maschine.

```
> data Instruction
>    = Unwind
>    | Pushglobal Name
>    | Pushint Int
>    | Push Int
>    | Mkap
>    | Update Int
>    | Pop Int
>    | Alloc Int
>    | Slide Int
>    | Eval
>    | Add | Sub | Mul | Div | Neg
>    | Eq | Ne | Lt | Le | Gt | Ge
>    | Pack Int Int
>    | Casejump (AssocList Int [Instruction])
>    | Split Int
>    | Print
>    | Ord
>    | Chr
```

Es gibt in der $G^{\#}$-Maschine Instruktionen, für die die G-Maschine keinen Bedarf hat. Dies sind:

- Die `Join` Anweisung kann mehrere alternative Ableitungsfolgen zusammenführen.

  ```
  >    | Join
  ```

- Um abstrakte Werte miteinander vergleichen zu können, hat die $G^{\#}$-Maschine die `AbsEq` Anweisung.

  ```
  >    | AbsEq
  ```

- Schließlich hat die $G^{\#}$-Maschine eine Instruktion um aus $n$ Knoten eine $n$-elementige Vereinigung zu machen.

  ```
  >    | MkUnion Int
  ```

Gelegentlich müssen Anweisungen miteinander verglichen werden, dazu machen wir aus dem Typ `Instruction` eine Instanz der Klasse `Eq`.

```
> instance Eq Instruction
>   where
>   Unwind      == Unwind       = True
>   Pushglobal a== Pushglobal b = a == b
>   Pushint a   == Pushint b    = a == b
>   Push a      == Push b       = a == b
>   Mkap        == Mkap         = True
>   Update a    == Update b     = a == b
>   Pop a       == Pop b        = a == b
>   Alloc a     == Alloc b      = a == b
>   Slide a     == Slide b      = a == b
>   Eval        == Eval         = True
>   Add         == Add          = True
>   Sub         == Sub          = True
>   Mul         == Mul          = True
>   Div         == Div          = True
>   Neg         == Neg          = True
>   Eq          == Eq           = True
>   Ne          == Ne           = True
>   Lt          == Lt           = True
>   Le          == Le           = True
>   Gt          == Gt           = True
>   Ge          == Ge           = True
>   Pack a b    == Pack c d     = a == c && b == d
```

```
>    Casejump a  == Casejump b   = a == b
>    Split a     == Split b      = a == b
>    Print       == Print        = True
>    Ord         == Ord          = True
>    Chr         == Chr          = True
>    Join        == Join         = True
>    AbsEq       == AbsEq        = True
>    MkUnion a   == MkUnion b    = a == b
>    _           == _            = False
```

Die Funktion `dispatch` entscheidet anhand der Anweisung welcher Zustandsübergang stattzufinden hat.

```
> dispatch :: Instruction -> GmState -> GmState
> dispatch (Pushglobal f)   = pushglobal f
> dispatch (Pushint n)      = pushint n
> dispatch Mkap             = mkap
> dispatch (Push n)         = push n
> dispatch (Update n)       = update n
> dispatch (Pop n)          = pop n
> dispatch Unwind           = unwind
> dispatch Eval             = geval
> dispatch (Alloc n)        = alloc n
> dispatch (Slide n)        = slide n
> dispatch Add              = gadd
> dispatch Sub              = gsub
> dispatch Mul              = gmul
> dispatch Div              = gdiv
> dispatch Neg              = gneg
> dispatch Eq               = geq
> dispatch Ne               = gne
> dispatch Lt               = glt
> dispatch Le               = gle
> dispatch Gt               = ggt
> dispatch Ge               = gge
> dispatch (Pack t n)       = pack t n
> dispatch (Casejump cs)    = casejump cs
> dispatch (Split n)        = split n
> dispatch Print            = gprint
> dispatch AbsEq            = gabseq
> dispatch (MkUnion n)      = mkunion n
> dispatch Ord              = gord
> dispatch Chr              = gchr
```

Die Zustandsübergänge selbst werden dann durch die folgenden Funktionen dargestellt.

Die Adresse eines globalen Symbols wird von `pushglobal` auf den Stapel gelegt.

```
> pushglobal :: Name -> GmState -> GmState
> pushglobal f state
>   = putStack (a: getStack state) state
>   where a = aLookup (getGlobals state) f
>           (error ("Undeclared global " ++ f))
```

Eine entsprechende Funktion ist `pushint`. Hier wird gegebenenfalls ein neues globales Symbol erzeugt und seine Adresse auf den Stapel gelegt. Wie bei allen neuen Knoten, sind erstmal beide Komponenten, die reduzierte und die ursprüngliche, gleich.

```
> pushint :: Int -> GmState -> GmState
> pushint n state
>   | aLookup gl (show n) 0 == 0
>       = putGlobals ((show n := addr):gl)
>           (putHeap newHeap (putStack (addr: s) state))
>   | otherwise = putStack ((aLookup gl (show n) 0): s) state
>   where   (newHeap, addr) = bAlloc (getHeap state) (NNum n)
>           gl      = getGlobals state
>           s       = getStack state
```

Mit `mkap` wird aus zwei Knoten ein Anwendungsknoten erzeugt, bei dem der Knoten an der Stapelspitze auf den darunterliegenden angewandt wird.

```
> mkap :: GmState -> GmState
> mkap state
>   = putHeap heap´ (putStack (a:as´) state)
>       where   (heap´, a)  = bAlloc (getHeap state) nd
>               nd = NAp a1 a2
>               (a1:a2:as´) = getStack state
```

Die `push` Anweisung kopiert uns das $n$-te Stapelelement an die Spitze.

```
> push :: Int -> GmState -> GmState
> push n state
>   = putStack ((as!!n):as) state
>   where   as = getStack state
```

### A.2.1 Update

Die `update` Anweisung überschreibt die Wurzel eines Graphen mit dem weiter abgeleiteten Graphen. Werden Vereinigungen reduziert, so soll nicht jedes der Elemente bis in WHNF reduziert werden, sondern für jedes Element soll nur ein Reduktionsschritt durchgeführt werden. Dazu wird die Wurzel eines solchen

Elements markiert. Findet `update` eine Wurzel markiert vor, wenn es sie überschreiben soll, dann wird eine `Join` Anweisung eingeschoben. Diese bewirkt, daß die Auswertung dieser Ableitung mit dem nächsten Schritt endet. Ebenso muß `update` einen Teil der Pfadanalyse übernehmen. Tritt in jedem Element einer Vereinigung ein Verweis auf die zu überschreibende Wurzel auf, dann kann die Stapelspitze durch `Bot` ersetzt werden. Natürlich sind alle offenen Ausdrücke, die nach der zu überschreibenden Wurzel eröffnet wurden dann auch nicht mehr offen.

```
> update :: Int -> GmState -> GmState
> update n state
>   | isMarked root = putCode (Join:cs) (putMarks (tail mks)
>       (putOpen op´ (putHeap hp´ (putStack as state))))
>   | otherwise = putOpen op´ (putHeap hp´ (putStack as state))
>   where
```

Muß `Bot` eingeführt werden?

```
>       newNode
>           | inEveryReductionPath hp a (lookup a) root
>               = hLookup hp (aLookup (getGlobals
>                   state) "Bot" (error
>                   "Undeclared global Bot"))
>           | otherwise = (oLookup hp root, NInd a)
```

Aus den offenen Redexen müssen ggf. alle Verweise auf die Wurzel entfernt werden.

```
>       op´
>           | (elem root op)
>               && (not (isRedex lookup (lookup a)))
>               = killInds (newOp (dropWhile (root /=) op)) root
>           | otherwise         = op
>       lookup = rLookup hp
>       root = as !! n
>       killInds [] _ = []
>       killInds (x:xs) y
>           | lookup x == (NInd y)  = killInds (killInds xs x) y
>           | otherwise             = x:(killInds xs y)
>       newOp [] = []
>       newOp (x:xs) = xs
>       hp = getHeap state
>       hp´= (hUpdate hp root newNode)
>       op = getOpen state
>       (a:as) = getStack state
>       cs = getCode state
>       isMarked a = elem a mks
>       mks = getMarks state
```

Betrachten wir die Funktion `inEveryReductionPath`, die entscheidet, ob der Graph ab dem 3. Argument einen Verweis auf die Adresse (4. Argument) in jedem Reduktionspfad enthält.

```
> inEveryReductionPath :: GmHeap -> Addr -> Node -> Addr -> Bool
```

Haben wir den zu untersuchenden (Teil) Graphen schon untersucht?

```
> inEveryReductionPath hp _ NVisited _ = False
```

Verweisen gilt es zu folgen, es sei denn, sie sind es wonach wir suchen.

```
> inEveryReductionPath hp a (NInd a´) root =
>   a´ == root
>   || inEveryReductionPath (rUpdate hp a NVisited) a´
>       (rLookup hp a´) root
```

Um in jedem Reduktionspfad einer Funktionsanwendung zu sein, genügt es, in jedem Reduktionspfad der Funktion *oder* des Argumentes zu sein.

```
> inEveryReductionPath hp a (NAp e1 e2) root =
>   inEveryReductionPath (update a) e1
>       (lookup e1) root
>   || inEveryReductionPath (update a) e2
>       (lookup e2) root
>   where
>       lookup = rLookup hp
>       update x = rUpdate hp x NVisited
```

Ein Verweis ist in jedem Reduktionspfad eines Konstruktors, wenn er in jedem Reduktionspfad *eines* seiner Argumente ist.

```
> inEveryReductionPath hp a (NConstr t as) root =
>   any (\x -> inEveryReductionPath (rUpdate hp a NVisited) x
>       (rLookup hp x) root) as
```

Um in jedem Reduktionspfad einer Vereinigung zu sein, ist es erforderlich, in jedem Reduktionspfad *aller* Elemente zu sein.

```
> inEveryReductionPath hp a (NUnion us) root =
>   all (\x -> inEveryReductionPath (rUpdate hp a NVisited) x
>       (rLookup hp x) root) (map fst us)
```

Sonst ist der Verweis nicht in jedem Reduktionspfad des Graphen.

```
> inEveryReductionPath _ _ g root = False
```

Die pop Anweisung nimmt $n$ Elemente vom Stapel.

```
> pop :: Int -> GmState -> GmState
> pop n state = putStack (drop n (getStack state)) state
```

alloc $n$ erzeugt $n$ Stapeleinträge, die alle auf mit `NNum 1` initialisierte Knoten zeigen.

```
> alloc :: Int -> GmState -> GmState
> alloc n state = putStack (alloced ++ (getStack state))
>           (putHeap newHeap state)
>   where (newHeap, alloced) = allocNodes n (getHeap state)
```

Um `alloc` zu implementieren verwenden wir `allocNodes`, die uns ein Paar bestehend aus einem neuen Heap und einer Liste der zugewiesenen Adressen liefert.

```
> allocNodes :: Int -> GmHeap -> (GmHeap, [Addr])
> allocNodes 0      heap    = (heap, [])
> allocNodes (n+1)  heap    = (heap2, a:as)
>   where
>       (heap1, as) = allocNodes n heap
>       (heap2, a)  = bAlloc heap1 (NNum 1)
```

Die `slide` Funktion entfernt $n$ unter der Stapelspitze liegende Elemente vom Stapel.

```
> slide :: Int -> GmState -> GmState
> slide n state
>   = putStack (a:drop n as) state
>    where (a:as) = getStack state
```

Trifft `dispatch` auf eine `Eval` Anweisung, wird `geval` aufgerufen. Dabei wird unterschieden, ob es sich um die abstrakten Werte `Top` oder `Bot` handelt, die nicht weiter ausgewertet werden dürfen oder ob es sich um einen anderen Wert handelt, der zu reduzieren ist.

```
> geval :: GmState -> GmState
> geval state
>   | (rLookup hp a) `elem` [NBot, NTop]    = state
>   | otherwise =
>       putCode [Unwind] (putStack [a]
>           (putDump ((i, s): getDump state) state))
>   where   (a:s)       = getStack state
>           i       = getCode state
>           hp      = getHeap state
```

### A.2.2 Arithmetische Anweisungen

Die Funktionen, die arithmetische Anweisungen realisieren, unterscheiden sich wesentlich von denen der konkreten G-Maschine.

```
> boxInteger :: Int -> GmState -> GmState
> boxInteger n state
>   = putStack (a: getStack state) (putHeap h´ state)
>           where (h´, a) = bAlloc (getHeap state) (NNum n)


> unboxInteger :: Addr -> GmState -> Int
> unboxInteger a state
>   = ub (rLookup (getHeap state) a)
>       where   ub (NNum i) = i
>               ub n        = error "Unboxing a non-integer"
```

Die `boxUnion` Funktion erzeugt aus einer entsprechenden Liste eine einfachte Vereinigung.

```
> boxUnion :: [(Addr, [Addr])] -> GmState -> GmState
> boxUnion us state
>   = putStack (a: getStack state)
>       (putHeap (simplify (rLookup h´) (rUpdate h´) h´ a) state)
>           where (h´, a) = bAlloc (getHeap state) (NUnion us)
```

Ein deutlicher Unterschied besteht darin, daß in der $G^{\#}$-Maschine auf die abstrakten Werte `Top` und `Union` eingegangen werden muß.

```
> primitive1 :: (b -> GmState -> GmState)   -- boxing function
>               -> (Addr -> GmState -> a)   -- unbixing function
>               -> (a -> b)                 -- operator
>               -> (GmState -> GmState)     -- state transition
> primitive1 box unbox op state = newstate n
>   where
>   (a:as) = getStack state
>   n = rLookup (getHeap state) a
>   newstate NTop = state
>   newstate (NUnion us)
>       = boxUnion us´ (putStack as state´)
>       where
>       (state´, us´) = mapAccuml trans state us
>       trans st (x, ops)
>           =   let p = primitive1 box unbox op
>                       (putStack (x:as) st)
>               in (p, (head (getStack p),ops))
>   newstate _ = box (op (unbox a state)) (putStack as state)
```

```
> primitive2 :: (b -> GmState -> GmState)   -- boxing function
>       -> (Addr -> GmState -> a)   -- unbixing function
>       -> (a -> a -> b)        -- operator
>       -> (GmState -> GmState)     -- state transition
> primitive2 box unbox op state = newstate n0 n1
>   where
>   (a0:a1:as) = getStack state
>   n0 = rLookup (getHeap state) a0
>   n1 = rLookup (getHeap state) a1
>   un = \x -> unbox x state
>   newstate NTop _ = putStack (a0:as) state
>   newstate _ NTop = putStack (a1:as) state
>   newstate (NUnion us) _
>       = boxUnion us´ (putStack as state´)
>       where
>       (state´, us´) = mapAccuml trans state us
>       trans st (x, ops)
>           =   let p = primitive2 box unbox op
>                       (putStack (x:a1:as) st)
>               in (p, (head (getStack p),ops))
>   newstate _ (NUnion us)
>       = boxUnion us´ (putStack as state´)
>       where
>       (state´, us´) = mapAccuml trans state us
>       trans st (x, ops)
>           =   let p = primitive2 box unbox op
>                       (putStack (a0:x:as) st)
>               in (p, (head (getStack p),ops))
>   newstate _ _    = box (op (un a0) (un a1))
>                       (putStack as state)


> arithmetic1 ::    (Int -> Int)        -- arithmetic operator
>           -> (GmState -> GmState) -- state transition
> arithmetic1 = primitive1 boxInteger unboxInteger


> arithmetic2 ::    (Int -> Int -> Int) -- arithmetic operation
>           -> (GmState -> GmState) -- state transition
> arithmetic2 = primitive2 boxInteger unboxInteger


> gadd :: GmState -> GmState
> gadd = arithmetic2 (+)


> gsub :: GmState -> GmState
> gsub = arithmetic2 (-)
```

```
> gdiv :: GmState -> GmState
> gdiv = arithmetic2 (div)


> gmul :: GmState -> GmState
> gmul = arithmetic2 (*)


> gneg :: GmState -> GmState
> gneg = arithmetic1 negate
```

### A.2.3 Vergleiche

Für die Vergleiche giilt ähnliches, wie für die arithmetischen Anweisungen.

```
> boxBoolean :: Bool -> GmState -> GmState
> boxBoolean b state =
>   putStack (a : getStack state) (putHeap h´ state)
>       where   (h´, a) = bAlloc (getHeap state) (NConstr b´ [])
>               b´  | b         = 2
>                   | otherwise = 1


> comparison :: (Int -> Int -> Bool) -> GmState -> GmState
> comparison = primitive2 boxBoolean unboxInteger


> geq :: GmState -> GmState
> geq = comparison (==)


> gne :: GmState -> GmState
> gne = comparison (/=)


> glt :: GmState -> GmState
> glt = comparison (<)


> gle :: GmState -> GmState
> gle = comparison (<=)


> ggt :: GmState -> GmState
> ggt = comparison (>)
```

```
> gge :: GmState -> GmState
> gge = comparison (>=)
```

Für die Verarbeitung von Zeichen benötigen wir zwei primitive Funktionen `ord` und `chr`, die es erlauben mit Zahlenwerten zu rechenen. Diese definieren wir mit Hilfe von `boxChar`, `unboxChar`, `boxInteger` und `unboxInteger`.

```
> boxChar :: Char -> GmState -> GmState
> boxChar  c state
>   = putStack (a: getStack state) (putHeap h´ state)
>   where
>   (h´, a) = bAlloc (getHeap state) (NConstr (ord c) [])


> unboxChar :: Addr -> GmState -> Char
> unboxChar a state
>   = ub (rLookup (getHeap state) a)
>       where   ub (NConstr c []) = chr c
>               ub n        = error "Unboxing a non-char"


> gord :: GmState -> GmState
> gord = primitive1 boxInteger unboxChar ord


> gchr :: GmState -> GmState
> gchr = primitive1 boxChar unboxInteger chr
```

Da die Funktion `==` strikt in ihren Argumenten ist, wir aber auch die abstrakten Werte `Top` und `Bot` miteinander vergleichen müssen, benötigen wir eine weitere Funktion: `gabseq`.

```
> gabseq :: GmState -> GmState
> gabseq state
>   = boxBoolean (neq hp (h a1) (h a2)) (putStack as state)
>   where   h = rLookup hp
>           hp = getHeap state
>           (a1:a2:as) = getStack state


> neq :: GmHeap -> Node -> Node -> Bool
> neq hp (NInd a) x = neq hp (rLookup hp a) x
> neq hp x (NInd a) = neq hp x (rLookup hp a)
> neq hp x y = x == y
```

### A.2.4 Primitive Funktionen

In diesem Abschnitt werden die primitiven Funktionen der $G^{\#}$-Maschine eingeführt, zusammen mit ihrer Striktheits-Information.

```
> module Primitives
>   (Primitives.., CompilerTypes..) where
> import CompilerTypes
```

Von den primitiven Funktionen sind alle bis auf `if` strikt in allen Argumenten. Um diese Striktheits-Information bei der Analyse nutzen zu können werden beim Aufruf die strikten Argumente ausgewertet. Kann eines der Argumente auf `Bot` reduziert werden, dann kann sofort `Bot` zurückgegeben werden.

```
> firstTwoStrictCode :: String -> GmCode
> firstTwoStrictCode f =
>   [Push 0,
>    Eval,
>    Pushglobal "Bot",
>    AbsEq,
>    Casejump [(2:=[Pushglobal "Bot",
>                   Update 3,
>                   Pop 3,
>                   Unwind]),
>              (1:=[Push 2,
>                   Eval,
>                   Pushglobal "Bot",
>                   AbsEq,
>                   Casejump [(2:= [Pushglobal "Bot",
>                                   Update 4,
>                                   Pop 4,
>                                   Unwind]),
>                             (1:= [Pop 2,
>                                   Pushglobal ('_':f),
>                                   Mkap,
>                                   Mkap,
>                                   Update 0,
>                                   Pop 0,
>                                   Unwind])]])]]


> oneStrictCode :: String -> GmCode
> oneStrictCode f =
>   [Push 0,
>    Eval,
>    Pushglobal "Bot",
>    AbsEq,
```

```
>    Casejump [(2:=[Pushglobal "Bot",
>                   Update 2,
>                   Pop 2,
>                   Unwind]),
>              (1:= [Pop 1,
>                    Pushglobal (´_´:f),
>                    Mkap,
>                    Update 0,
>                    Pop 0,
>                    Unwind])]]
```

```
> compiledPrimitives :: [GmCompiledSC]
> compiledPrimitives =
>      [("+", 2, firstTwoStrictCode "+"),
>   ("-", 2, firstTwoStrictCode "-"),
>   ("*", 2, firstTwoStrictCode "*"),
>   ("/", 2, firstTwoStrictCode "/"),
>   ("_+", 2, compiledBinaryPrimitiveCode Add),
>   ("_-", 2, compiledBinaryPrimitiveCode Sub),
>   ("_*", 2, compiledBinaryPrimitiveCode Mul),
>   ("_/", 2, compiledBinaryPrimitiveCode Div),
>
>   ("_negate", 1, compiledUnaryPrimitiveCode Neg),
>   ("negate", 1, oneStrictCode "negate"),
```

Die Vergleiche werden dann folgendermaßen implementiert.

```
>   ("==", 2, firstTwoStrictCode "=="),
>   ("~=", 2, firstTwoStrictCode "~="),
>   ("<", 2, firstTwoStrictCode "<"),
>   ("<=", 2, firstTwoStrictCode "<="),
>   (">", 2, firstTwoStrictCode ">"),
>   (">=", 2, firstTwoStrictCode ">="),
>   ("_==", 2, compiledBinaryPrimitiveCode Eq),
>   ("_~=", 2, compiledBinaryPrimitiveCode Ne),
>   ("_<", 2, compiledBinaryPrimitiveCode Lt),
>   ("_<=", 2, compiledBinaryPrimitiveCode Le),
>   ("_>", 2, compiledBinaryPrimitiveCode Gt),
>   ("_>=", 2, compiledBinaryPrimitiveCode Ge),
```

Die Primitiven für die Zeichenmanipulation werden hier implementiert:

```
>   ("_ord", 1, compiledUnaryPrimitiveCode Ord),
>   ("_chr", 1, compiledUnaryPrimitiveCode Chr),
>   ("ord", 1, oneStrictCode "ord"),
```

```
>   ("chr", 1, oneStrictCode "chr"),
>   ("strict", 2,
>       [Push 1,
>        Eval,
>        Pushglobal "Bot",
>        AbsEq,
>        Casejump [(2 :=    [Pushglobal "Bot"]),
>                  (1 :=    [Push 2,
>                             Push 2,
>                             Mkap])],
>        Update 3,
>        Pop 3,
>        Unwind])]

> compiledUnaryPrimitiveCode :: Instruction -> GmCode
> compiledUnaryPrimitiveCode instr
>   = [ Push 0,
>       Eval,
>       instr,
>       Update 1,
>       Pop 1,
>       Unwind]


> compiledBinaryPrimitiveCode :: Instruction -> GmCode
> compiledBinaryPrimitiveCode instr
>   = [ Push 1, Eval,
>       Push 1, Eval,
>       instr,
>       Update 2, Pop 2, Unwind]
```

### A.2.5 Datenstrukturen

```
> pack :: Int -> Int -> GmState -> GmState
> pack t n state
>   = putHeap heap´ (putStack (a:as) state)
>   where
>       (heap´, a) = bAlloc (getHeap state) (NConstr t a1n)
>       (a1n, as) = splitAt n (getStack state)


> casejump :: AssocList Int GmCode -> GmState -> GmState
> casejump cs state
>   = casealts (rLookup (getHeap state) a) cs state
>   where   (a : as) = getStack state
```

```
> casealts :: Node -> AssocList Int GmCode -> GmState -> GmState
> casealts (NInd a) cs state
>   = casealts (rLookup (getHeap state) a) cs state
> casealts (NConstr t ss) cs state
>   = putCode (aLookup cs t [Unwind] ++ i) state
>   where   i = getCode state
```

```
> casealts NTop cs state
```

= putHeap h"' (putPaths (PList l) (putStack (addr:ss) newstate))

```
>   = putHeap h´´´
>                  (putStack (addr:ss) newstate)
>   where
>       l =         init (evalPaths [[state]] [b|(a := b) <- cs])
>       newstate =  last (head l)
>       (s:ss) =    getStack newstate
>       (h´, addr) =bAlloc (getHeap newstate)
>              (NUnion [(head (getStack s´´), getOpen s´´)
>                  | s´ <- l, s´´ <- [last s´]])
>       h´´ = simplify (rLookup h´) (rUpdate h´) h´ addr
>       h´´´= simplify (oLookup h´´) (oUpdate h´´) h´´ addr
```

casealts (NUnion us) cs state = putHeap h"' (putPaths (PList l) (putStack (addr:ss) newstate))

```
> casealts (NUnion us) cs state = putHeap h´´´
>              (putStack (addr:ss) newstate)
>   where   l = init (unioncase [[state]] us cs (getCode state)
>                  (if (or [isRedex (rLookup hp) (rLookup hp tst)|(tst,_) <- us])
>                      then [Eval, Join, Casejump cs]
>                      else [Casejump cs, Join]))
>           hp = getHeap state
>           newstate = last (head l)
>           (s:ss) = getStack newstate
>           (h´, addr) = bAlloc (getHeap newstate)
>                  (NUnion [(head (getStack s´´), getOpen s´´)
>                      |s´ <- l, s´´ <- [last s´]])
>           h´´ = simplify (rLookup h´) (rUpdate h´) h´ addr
>           h´´´= simplify (oLookup h´´) (oUpdate h´´) h´´ addr
```

```
> casealts NBot cs state = state
```

Für den Fall das `Top` auf dem Stapel liegt sollen alle Variablen an `Top` gebunden werden. Dazu kopieren wir $n$ `Tops` auf den Stapel.

```
> split :: Int -> GmState -> GmState
> split n state = putStack (bindem (rLookup hp a) ++ ss) state
>   where
>       bindem (NConstr t as)   = as
>       bindem NTop         = take n (repeat a)
>       bindem (NInd b)     = bindem (rLookup hp b)
>       hp = getHeap state
>       (a : ss) = getStack state
```

Durch Matching mit `Top` können alternative Ableitungen gestartet werden. Dies wird dargestellt durch jeweils einen Aufruf von `eval`, wobei nach den Anweisungen der Alternative eine `Join` Anweisung eingeschoben wird. Wir benötigen am Ende einen Heap in dem alle möglichen Ergebnisse gültig sind, deswegen wird der Heap durch die Auswertungen durchgereicht.

```
> evalPaths :: [[GmState]] -> [GmCode] -> [[GmState]]
> evalPaths s [] = s
> evalPaths (s:ss) (c:cs) = evalPaths (s´:s:ss) cs
>   where
>       s´ = eval (putStack stack
>                   (putCode (c++[Join]++i)
>                       l))
>       i = getCode l
>       stack = getStack (head s)
>       l = last s
```

Wenn mit `Unions` gematcht wird, dann sollen die Variablen gemäß allen Matches jedes Elementes gebunden werden. Es kann also vorkommen, daß eine Anweisungsfolge mit verschiedenen Elementen der `Union` durchgeführt wird.

```
> unioncase :: [[GmState]] -> [(Addr, [Addr])]
>       -> AssocList Int GmCode -> GmCode -> GmCode -> [[GmState]]
> unioncase s [] cs i ev            = s
> unioncase (s:ss) ((u1, u2):us) cs i ev    = unioncase (s´:s:ss) us cs i ev
>   where
```

i = getCode l

```
>       s´ = eval (putStack (u1:as) (putOpen u2
>                   (putCode (ev++i) l)))
```

(putCode ((head i):Join:(tail i)) l)))

```
>       (a:as) = getStack (head s)
>       l = last s
```

### A.2.6 Ausgabe

```
> gprint :: GmState -> GmState
> gprint state = printNode state node
>   where
>       node = rLookup (getHeap state) a
>       (a : as) = getStack state


> printNode :: GmState -> Node -> GmState
> printNode state n
>   = putOutput (iConcat [op, delim, showNode state a n])
>       (putStack as state)
>   where
>       (a : as) = getStack state
>       op = getOutput state
>       delim
>           | op /= iNil    = iStr ", "
>           | otherwise = iNil
```

Mit `mkunion` bildet man aus den auf dem Stapel liegenden Elementen eine Vereinigung. Vereinigungen sollen immer vereinfacht sein.

```
> mkunion :: Int -> GmState -> GmState
> mkunion n state = putStack (addr´:(drop n as))
>                       (putHeap h´´ state)
>   where
>       as = getStack state
>       op = getOpen state
>       (h´, addr´) = bAlloc (getHeap state)
>           (NUnion (take n (zip as (repeat op))))
>       h´´ = simplify (rLookup h´) (rUpdate h´) h´ addr´
>       h´´´= simplify (oLookup h´´) (oUpdate h´´) h´´ addr´
```

### A.2.7 Unwind

```
> unwind :: GmState -> GmState
> unwind state = newState (rLookup heap a)
>   where
>       a´              = aLookup (getGlobals state) "Bot"
>                           (error "Can´t find Bot")
>       b´              = aLookup (getGlobals state) "Top"
>                           (error "Can´t find Top")
>       (a:as)          = getStack state
>       heap            = getHeap state
```

```
>       dp              = getDump state
>       ((i´,s´):d´)    = dp
>       cd              = getCode state
>       op              = getOpen state
>       hp              = rLookup heap
```

Als Default wird auf den Dump zugegriffen.

```
>       std | dp == []  = state
>           | otherwise = putCode (cd ++ i´)
>                             (putStack (a:(as++s´))
>                                 (putDump d´ state))
>       absStd
>           | dp == []  = state
>           | otherwise = putCode (cd ++ i´)
>                             (putStack (a:s´)
>                                 (putDump d´ state))

>       newState (NNum n)       = std
>       newState (NConstr t as) = std
>       newState NTop           = absStd
>       newState NBot           = absStd

>       newState x@(NAp a1 a2)
>           | hit /= [] = putHeap h´ (putCode
>               (if (elem a (getMarks state))
>                   then []
>                   else [Unwind]) state)
>           | isRedex hp x = putOpen (a:op) (putCode (Unwind : cd)
>                   (putStack (a1:a:as) state))
>           | otherwise = putCode (Unwind : cd)
>                   (putStack (a1:a:as) state)
>           where
>               h´  = rUpdate heap a (NInd (head hit))
>               hit = [x | x <- op, a == x
>                       || lessEqual heap (rLookup heap a)
>                           (oLookup heap x) []]

>       newState (NGlobal n c)
>           | (length as) >= n = putCode (c ++ cd)
>                       (putStack rs state)
>           | otherwise    = putCode (cd ++ i´)
>                       (putStack (a:(as++s´))
>                           (putDump d´ state))
>           where rs = rearrange n heap (a:as)
```

```
>       newState (NInd a1)
>           | elem a1 op = putCode (Unwind : cd)
>                             (putStack (a´:as) state)
>           | otherwise = putCode (Unwind : cd)
>                             (putStack (a1:as) state)

>       newState x@(NUnion u)
>           | any (getOpenTermLimit state <=) [length ops|(_,ops) <- u]
>               = putStack (b´:a:as) (putCode [Update 0] state)
>           | isRedex hp x
```

= putHeap newHeap' (putPaths (PList l) (putCode [Unwind] state))

```
>               = putHeap newHeap´
>                   (putCode [Unwind] state)
>
>           | otherwise = std
>           where
>           l = init (rewriteEach [[state]] (Unwind:cd) u)
>           lastState = last (head l)
>           newHeap = rUpdate (getHeap lastState) a
>               (NUnion (zip (map fst u)
>                   (map (getOpen . last) (reverse l))))
>           newHeap´ = simplify (rLookup newHeap)
>               (rUpdate newHeap) newHeap a

> isRedex :: (Addr -> Node) -> Node -> Bool
> isRedex _ (NNum n)    = False
> isRedex _ (NConstr t as)= False
> isRedex _ NTop    = False
> isRedex _ NBot    = False
> isRedex f (NInd a)    = isRedex f (f a)
> isRedex f (NUnion us) = all ((isRedex f) . f . fst) us
> isRedex f x@(NAp a1 a2) = isSaturated g (length spine)
>   where   (g:spine) = collectSpine f x
>           collectSpine f (NAp a1 a2) = collectSpine f (f a1)
>                                           ++ [f a2]
>           collectSpine f (NInd a)    = collectSpine f (f a)
>           collectSpine f nd          = [nd]
>           isSaturated (NGlobal n cs) m = m >= n
>           isSaturated NTop n          = False
>           isSaturated NBot n          = False
> isRedex _ _       = False
```

Die Vereinigungen müssen für `lessEqual` bereits vereinfacht sein.

Die `lessEqual` Funktion approximiert die $\leq$ Relation auf abstrakten Graphen. Sie wird für die Vereinfachung von Vereinigungen und für Reduktionspfad-Analyse verwendet.

```
> lessEqual :: GmHeap -> Node -> Node -> [(Node, Node)] -> Bool
> lessEqual _   NBot     _     _ = True
> lessEqual _   _    NTop      _ = True
> lessEqual hp  (NInd a) b     p = a´ == b || lessEqual hp a´ b p
>   where   a´ = rLookup hp a
> lessEqual hp  a    (NInd b) p = a == b´ || lessEqual hp a b´ p
>   where   b´ = oLookup hp b

> lessEqual hp  x@(NAp d e) y@(NAp f g) p =
>   (d == f || lessEqual hp (h d) (h´ f) p´) &&
>   (e == g || lessEqual hp (h e) (h´ g) p´)
>   where   p´ = (x, y) : p
>           h = rLookup hp
>           h´= oLookup hp

> lessEqual hp a@(NConstr t f)  b@(NConstr u g) p =
>   t == u
>   && all (\ (x, y) -> lessEqual hp (h x) (h´ y) p´) (zip f g)
>   where   p´ = (a, b) : p
>           h = rLookup hp
>           h´= oLookup hp

> lessEqual h   t       y@(NUnion us)   p = or (map leq us)
>   where leq x = lessEqual h t (oLookup h (fst x)) ((t, y) : p)

> lessEqual hp  (NUnion us) t´      p =
>   all (\ x -> lessEqual hp (rLookup hp (fst x)) t´ p) us

> lessEqual _   t       t´      p = t == t´ || elem (t, t´) p

> rewriteEach :: [[GmState]] -> GmCode -> [(Addr, [Addr])]
>                   -> [[GmState]]
> rewriteEach s [] _ = s
> rewriteEach s _ [] = s
> rewriteEach (s:ss) cs (u:us) = rewriteEach (s´:s:ss) cs us
>   where   s´ = eval (putCode cs (putStack ((fst u):sts)
>                       (putOpen (snd u)
>                           (putMarks [a] (last s)))))
>           a  = fst u
>           (st:sts) = getStack (head s)
```

### A.2.8 Vereinfachen von Vereinigungen

Wir benötigen die Maxima aus einer Vereinigung, um diese zu vereinfachen. Also wollen wir alle Elemente $x$ behalten, zu denen wir in $us$ kein $y$ finden können, so daß $x \leq y$. `maxima` setzt dies um.

```
> maxima :: (a -> a -> Bool) -> [a] -> [a] -> [a]
> maxima _ [] ys= ys
> maxima leq (x:xs) ys
>   | (any lx xs) || (any lx ys)
>       = maxima leq xs ys
>   | otherwise
>       = maxima leq xs (x:ys)
>   where lx = leq x
```

Um Vereinigungen zu vereinfachen, müssen wir Selbst-Referenzierung, Singletons und kleine Elemente entfernen. `simplify` erwartet eine `NUnion`-Adresse als viertes Argument. Außerdem werden die Vereinigungen vor der Vereinigung flach gemacht. Dabei reicht es aus `flat` end-rekursiv zu gestalten, denn Vereinigungen werden immer vereinfacht sobald sie erzeugt sind.

```
> simplify :: (Addr -> Node) -> (Addr -> Node -> GmHeap)
>       -> GmHeap -> Addr -> GmHeap
> simplify look upd h a = upd a new
>   where
>       new
>           | null vs   = hp v
>           | otherwise = NUnion (sort (v:vs))
>       (v:vs)  = maxima (\x y ->
>               lessEqual h (hp x) (hp y) []) ws []
>       flat [] = []
>       flat ((x,ops):xs) = (n (look x)) ++ (flat xs)
>           where   n (NUnion cs) = cs
>                   n _           = [(x,ops)]
>       ws = [b| b <- flat us, fst b /= a]
>       hp = look . fst
>       (NUnion us) = look a
```

Um sicherzustellen, daß der Algorithmus terminiert, müssen wir einige Resourcen beschränken. Hier wird die Zahl der offenen Auswertungen beschränkt.

openTermLimit :: Int openTermLimit = 10

```
> resourcesExhausted :: GmState -> Bool
> resourcesExhausted state = length (getOpen state)
>                               > getOpenTermLimit state
```

Wenn also ein Zustand erreicht wird, in dem die Länge der Liste von offenen Auswertungen `openTermLimit` übersteigt, dann wird der Reduktionspfad, der zu diesem Zustand geführt hat als erfolglos angesehen.

## A.3 Striktheits-Analyse

Um Striktheit im $i$-ten Argument einer Funktion zu analysieren, beginnen wir eine Reduktion, wobei alle Argumente der Funktion auf $\top$ gesetzt sind und das $i$-te Argument auf $\top$ gesetzt sind und das $i$-te Argument auf $\perp$. Diese Reduktion kann unsere Resourcenbeschränkung überschreiten. In diesem Fall können wir eine weitere Reduktion beginnen, wobei diesmal alle Argumente auf $\top$ gesetzt werden. Wenn keine dieser beiden Reduktionen $\perp$ innerhalb der Resourcenbeschränkung liefert, nehmen wir, um sicher zu gehen, an, daß die Funktion im $i$-ten Argument nicht strikt ist.

Hier ist also eine Funktion, die aus einem Zustand, einer Argumentposition und dem Namen einer Funktion eine Liste von Anfangszuständen macht. Diese Anfangszustände werden dann jeweils an `eval` zur Auswertung übergeben. Wenn `eval` einen Zustand liefert, der unsere Resourcenbeschränkung erfüllt und an dessen Stapelspitze $\perp$ steht, dann ist die Funktion strikt in ihrem $i$-ten Argument. Den Zustand muß man hier benutzen, damit Striktheits-Information, die bereits ermittelt wurde, verwendet werden kann.

```
> analyserStartStates :: GmState -> Name -> Int -> [GmState]
> analyserStartStates state f i
>   = [putCode (ins "Bot") state] -- , putCode (ins "Top") state]
>   where
>   (NGlobal n _) = rLookup (getHeap state)
>                         (aLookup (getGlobals state) f
>                         (error ("Undefined Global " ++ f)))
>   ins x = take (n-i) (repeat (Pushglobal "Top")) ++
>           [Pushglobal x] ++
>           take (i-1) (repeat (Pushglobal "Top")) ++
>           [Pushglobal f] ++
>           take n (repeat Mkap) ++
>           [Eval]
```

Haben wir eine Funktion untersucht, dann müssen wir die gewonnene Information speichern, um sie bei der Analyse weiterer Funktionen verwenden zu können. Dazu verwenden wir einen Trick, der die Grundidee der G-Mashine aufgreift. Wir kompilieren die Striktheits-Information zu G-Code, der, wenn er ausgeführt wird, die Anwendung der Striktheits-Information bewirkt. Die Striktheis-Information anzuwenden bedeutet für die Analyse, wenn an einer als strikt erkannten Position der Wert $\perp$ erscheint, als Wert der Funktion $\perp$ zu liefern. Für die strikten Positionen wird also folgendes getan: das Argument wird ausgewertet und wenn es $\perp$ ergibt wird sofort zurückgekehrt mit $\perp$ an der Stapelspitze. Findet sich an keiner der strikten Positionen $\perp$, so wird die ursprüngliche Funktion aufgerufen. Das Aktualisieren der Wurzeln beim Auswerten stellt sicher, daß die Argumente im Körper der ursprünglichen Funktion nicht erneut ausgewertet werden. (Genauer: das zur Auswertung nur noch nach dem Wert an der Wurzel geschaut werden muß.) Gegenüber der Auswertungszeit ohne Berücksichtigung von Striktheits-Information haben wir also nur wenig mehr Aufwand.

```
> analyserCode :: Int -> [Bool] -> Name -> GmCode
> analyserCode arg [] f =   [Pushglobal ('_':f)] ++
>               take arg (repeat Mkap) ++
>               [Update 0, Unwind]
```

Alternativ, untersuchen wir, ob der Aufruf der ursprünglichen Funktion notwendig ist oder ob es genügt einfach $\top$ zurückzugeben.

```
 analyserCode arg [] f =    [Pushglobal "Top", Update arg,
                Pop arg, Unwind]
```

```
> analyserCode arg (False:xs) f = analyserCode (arg+1) xs f
> analyserCode arg (True:xs) f  =
>   [Push arg,
>    Eval,
>    Pushglobal "Bot",
>    AbsEq,
>    Casejump [(2:= [Pushglobal "Bot",
>                    Update (arg+2+length xs),
>                    Pop (arg+2+length xs),
>                    Unwind]),
>              (1:= ((Pop 1):(analyserCode (arg+1) xs f)))]]
```

`strictPosition` ist die Funktion, die die Liste der Anfangszustände als Eingabe erhält und diese ihrerseits zur Auswertung weitergibt und prüft, ob das Resultat $\perp$ ist und ob es innerhalb unserer Resourcenbeschränkung ermittelt wurde.

```
> strictPosition :: [GmState] -> [Bool]
> strictPosition [] = []
> strictPosition (x:xs) =
>   let e = last (eval x); hp = getHeap e in
>       [not (resourcesExhausted e) &&
>       (neq hp (rLookup hp (head (getStack e))) NBot)]
>       ++ strictPosition xs
```

`strictPositions` wendet `strictPosition` auf alle Argumentpositionen einer Funktion an.

```
> strictPositions :: GmState -> Name -> [Bool]
> strictPositions state f =
>   map (\z -> any (True ==)
>       (strictPosition (analyserStartStates state f z))) [1..n]
>   where   (NGlobal n _) = rLookup (getHeap state)
>                  (aLookup (getGlobals state) f
>                   (error ("Undefined Global " ++ f)))
```

`analyseFunction` analysiert *eine* Funktion, deren Name ihr übergeben wird.

```
> analyseFunction :: Name -> GmState -> GmState
> analyseFunction f state =
>   putGlobals ((('_':f) := olda):(aUpdate gl f newa))
>       (putHeap newh
>       (putOutput
>           (iConcat [op, delim,
>                 iStr f, iStr " [",
>                 iInterleave (iStr ", ")
>                   (map (iStr . myshow)  striP),
>                 iStr "]"])
>           state))
>   where   (newh, newa) = bAlloc hp (NGlobal n
>                               (analyserCode 0 striP f))
>           olda = aLookup gl f
>                       (error ("Undefined Global " ++ f))
>           (NGlobal n _) = rLookup hp olda
>           hp = getHeap state
>           gl = getGlobals state
>           striP = strictPositions state f
>           op = getOutput state
>           delim
>               | op /= iNil = iNewline
>               | otherwise  = iNil
>           myshow True = "strict"
>           myshow False = "?"
```

`analyseFunctions` analysiert *alle* Funktionen, deren Namen in einer Liste übergeben werden.

```
> analyseFunctions :: [Name] -> GmState -> GmState
> analyseFunctions [] state = state
> analyseFunctions (x:xs) state
>   = analyseFunctions xs st
>   where
>       st = analyseFunction x state --(putStack [] (putDump []
> --            (putOpen [] (putMarks [] (putCode [] state))))
```

Um alle Funktionen in einer Datei zu analysieren verwenden wir die Funktion `allNames`, die uns eine Liste der Namen aller Funktionen liefert.

```
> allNames :: GmState -> [Name]
> allNames st = [nm | (nm := a) <- getGlobals st,
>           (NGlobal _ _) <- [rLookup (getHeap st) a]]
```

## A.4 Abhängigkeitsanalyse

Um die Abhängigkeit zwischen Funktionen zu untersuchen, benötigen wir ein Mittel, um herauszufinden, welche Funktionen von einem bestimmten Codefragment aufgerufen werden.

```
> calledNamesFromCode :: GmCode -> [Name]
> calledNamesFromCode (Pushglobal f : xs)
>   = f : calledNamesFromCode xs
> calledNamesFromCode (Casejump cs : xs)
>   = (concat [calledNamesFromCode x | _ := x <- cs])
>       ++ calledNamesFromCode xs
> calledNamesFromCode (_:xs) = calledNamesFromCode xs
> calledNamesFromCode []     = []
```

```
> calledNames :: Node -> [Name]
> calledNames (NGlobal n cs)    = calledNamesFromCode cs
> calledNames _         = []
```

Hier folgen einige Funktionen die sich auf die Abhängigkeitsuntersuchung beziehen.

Wir benötigen außerdem Typen für gerichtete Graphen und Funktionen, die wir beim Besuch eines Knotens ausführen. Betrachten wir zunächst die Graphen. Wir verwenden `CallGraph`en, Graphen, die die Aufruf-Struktur des Programms wiedergeben sollen.

```
> data CallGraphEdge = Name :=> Name
> type CallGraph = (AssocList Name (Bool, Int), [CallGraphEdge])
> type SCC = [Name]
```

Wir definieren eine Ordnung auf den Graphen, anhand der Nummern der Knoten. In der Haskell Prelude sind die folgenden Instanzen bereits vorhanden, daher sind sie nur in der Gofer Version im Text.

Aus einem Zustand können wir dann mit `callGraph` den Graphen der Aufruf-Relation erzeugen.

```
> callGraph :: GmState -> CallGraph
> callGraph s = (v, e)
>   where
>       v = [n := (False,0)|(n := _) <- glob]
>       e = [x :=> y| (x := a) <- glob,
>               y <- nub (calledNames (rLookup (getHeap s) a))]
>       glob = getGlobals s
```

Einen Graphen umzudrehen bedeutet dann, Quelle und Ziel der gerichteten Kanten zu vertauschen.

```
> reverseGraph :: CallGraph -> CallGraph
> reverseGraph (v,e) = (v,[y:=>x|x:=>y <- e])
```

Die starken Zusammenhangskomponenten finden wir dann mit DFS vom größten verbleibenden Knoten.

```
> findSCCs :: CallGraph -> [SCC] -> [SCC]
> findSCCs ([],_) s = s
> findSCCs (v,e)  s = findSCCs g (sc:s)
>   where
>       (n := _) = maximum v
>       (sc, g) = dfsVisitEarly n (v,e) ([],(v,e))
```

Wobei `dfs` hier nur eine Liste von Knoten, die per DFS gefunden wurden liefert. Unterscheiden wir zwischen `dfsVisitEarly` und `dfsVisitLate`, im einen Fall werden die Knoten in der Reihenfolge gesammelt, in der sie aufgefunden werden, im anderen Fall sammeln wir einen Knoten erst ein, wenn alle seine Nachfolger eingesammelt sind.

```
> dfsVisitEarly :: Name -> CallGraph -> ([Name], CallGraph)
>           -> ([Name], CallGraph)
> dfsVisitEarly n (v,e) (sc,r) =
>   dfss dfsVisitEarly [y | (x :=> y) <- e´, x == n] (v´,e´)
>       (n:sc, (v´,e´))
>   where
>       v´ = [n´ := (x,y)|(n´ := (x,y)) <- v, n´ /= n]
>       e´ = [x :=> y|(x :=> y) <- e, y /= n]


> dfsVisitLate :: Name -> CallGraph -> ([Name], CallGraph)
>           -> ([Name], CallGraph)
> dfsVisitLate n (v,e) (s,r) = (n:sc, res)
>   where
>   (sc, res) = dfss dfsVisitLate [y | (x :=> y) <- e´, x == n]
>       (v´,e´) (s,(v´,e´))
>   v´ = [n´ := (x,y)|(n´ := (x,y)) <- v, n´ /= n]
>   e´ = [x :=> y|(x :=> y) <- e, y /= n]
```

```
> dfss :: ( Name -> CallGraph -> ([Name], CallGraph)
>           -> ([Name], CallGraph)) -> [Name] -> CallGraph
>           -> ([Name], CallGraph) -> ([Name], CallGraph)
> dfss dfs [] _ r = r
> dfss dfs (n:ns) g r = dfss dfs ns r´ (sc´,r´)
>   where
>       (sc´,r´) = dfs n g r


> sccNumbering :: [Name] -> Int -> [(Name, Int)]
> sccNumbering [] _ = []
> sccNumbering (n:ns) i = (n,i) : (sccNumbering ns (i-1))


> callGraphSCCs :: CallGraph -> [SCC]
> callGraphSCCs callGraph@(v,e) =
>   findSCCs (reverseGraph ([n := (b,i)|(n := (b,i)) <- v´,
>                i /= 0, n /= "Top", n /= "Bot",
>                aLookup v´ (´_´:n) (False,0) == (False,0)],e)) []
>   where
>   (nameList,_) = dfsVisitLate "main" callGraph ([], callGraph)
>   v´ =  foldl (\x (y,z) -> aUpdate x y (False, z)) v
>           (sccNumbering nameList (length nameList))
```

# Anhang B

# Beispiel-Analysen

Die in diesem Anhang analysierten Funktionen stammen aus dem Programmtext der Striktheits-Analyse selbst, sowie aus der `standard.prelude`. Verwendet wurde eine Datei mit 287 Funktionen, in der:

- der Compiler vom Sprachkern zu $G^{\#}$-Code
- der Lexer für den Sprachkern
- einige Hilfsfunktionen
- die `standard.prelude` des Gofer-Interpreters mit Ausnahme der Fließkomma-Funktionen und der Typklassen

enthalten sind.

In der folgenden Liste steht die ermittelte Striktheits-Information neben dem Namen des Superkombinators und für jedes Argument wird `strict` oder `?` eingetragen. `strict` bedeutet dabei, daß die Analyse bei diesem Argument Striktheit feststellte und `?` bedeutet, daß keine Striktheit festgestellt werden konnte.

```
I [strict]
K [strict, ?]
K1 [?, strict]
S [strict, ?, ?]
compose [strict, ?, ?]
twice [strict, ?]
if [strict, ?, ?]
help []
const [strict, ?]
id [strict]
curry [strict, ?, ?]
uncurry [strict, strict]
fst [strict]
```

```
snd [strict]
fst3 [strict]
snd3 [strict]
thd3 [strict]
flip [strict, ?, ?]
binand [strict, ?]
binor [strict, ?]
not [strict]
and [strict]
or [strict]
any [?, strict]
all [?, strict]
otherwise []
isAscii [strict]
isControl [strict]
isPrint [strict]
isSpace [strict]
isUpper [strict]
isLower [strict]
isAlpha [strict]
isDigit [strict]
isAlphanum [strict]
toUpper [strict]
toLower [strict]
minChar []
maxChar []
even [strict]
odd [strict]
rem [strict, strict]
gcd´ [strict, strict]
gcd [strict, strict]
lcm [?, strict]
power [?, strict]
power´ [?, strict, ?]
power´´ [?, strict, ?]
abs [strict]
signum [strict]
plus [strict, strict]
minus [strict, strict]
mul [strict, strict]
sum [strict]
product [strict]
sums [strict]
products [strict]
head [strict]
last [strict]
tail [strict]
```

```
init [strict]
append [strict, ?]
length [strict]
lam_length [strict, ?]
at [strict, strict]
iterate [?, ?]
repeat [?]
cycle [strict]
copy [strict, ?]
nub [strict]
reverse [strict]
eq [strict, strict]
elem [?, strict]
notElem [?, strict]
max [strict, strict]
min [strict, strict]
maximum [strict]
minimum [strict]
concat [strict]
cons22 [?, ?]
transpose [strict]
lam_transpose [strict, ?]
null [strict]
delone [strict, strict]
del´ [strict, ?]
map [?, strict]
filter [?, strict]
foldl [?, ?, strict]
foldl1 [?, strict]
foldl´ [?, ?, strict]
scanl [?, ?, strict]
scanl1 [?, strict]
scanl´ [?, ?, strict]
foldr [?, ?, strict]
foldr1 [?, strict]
scanr [?, ?, strict]
scanr1 [?, strict]
take [strict, ?]
drop [strict, strict]
lam_splitAt [?, strict]
splitAt [strict, ?]
takeWhile [?, strict]
takeUntil [?, strict]
dropWhile [?, strict]
span [?, strict]
break [?, strict]
lines [strict]
```

```
words [strict]
unlines [strict]
lam_unlines [strict]
unwords [strict]
lam_unwords [strict, ?]
merge [strict, strict]
sort [strict]
insert [?, strict]
gt [strict, strict]
le [strict, strict]
qsort [strict]
zip [strict, ?]
lam_zip [?, ?]
zip3 [strict, ?, ?]
lam_zip3 [?, ?, ?]
zip4 [strict, ?, ?, ?]
lam_zip4 [?, ?, ?, ?]
zip5 [strict, ?, ?, ?, ?]
lam_zip5 [?, ?, ?, ?, ?]
zip6 [strict, ?, ?, ?, ?, ?]
lam_zip6 [?, ?, ?, ?, ?, ?]
zip7 [strict, ?, ?, ?, ?, ?, ?]
lam_zip7 [?, ?, ?, ?, ?, ?, ?]
zipWith [?, strict, ?]
zipWith3 [?, strict, ?, ?]
zipWith4 [?, strict, ?, ?, ?]
zipWith5 [?, strict, ?, ?, ?, ?]
zipWith6 [?, strict, ?, ?, ?, ?, ?]
zipWith7 [?, strict, ?, ?, ?, ?, ?, ?]
unzip [strict]
show [strict]
cjustify [strict, strict]
ljustify [?, strict]
rjustify [strict, strict]
space [strict]
layn [strict]
lay [?, strict]
main []
enumFrom [?]
until [strict, ?, ?]
until´ [strict, ?, ?]
error [strict]
hInitial []
hAlloc [strict, ?]
remove [strict, strict]
hUpdate [strict, ?, ?]
hFree [strict, ?]
```

```
aLookup [strict, ?, ?]
showaddr [?]
hLookup [strict, ?]
I2975_hAddresses [strict]
hAddresses [strict]
hSize [strict]
hNull []
hIsnull [strict]
I2978_aDomain [strict]
aDomain [strict]
I2981_aRange [strict]
aRange [strict]
aEmpty []
I2984_aDelete [?, strict]
aDelete [strict, ?]
aUpdate [?, ?, ?]
initialNameSupply []
makeName [strict, ?]
getName [?, ?]
getNames []
setEmpty []
setIsEmpty [strict]
setSingleton [?]
rmdup [strict]
setFromList [strict]
setToList [strict]
setUnion [strict, strict]
setIntersection [strict, strict]
setSubtraction [strict, strict]
setElementOf [?, strict]
setUnionList [strict]
first [strict]
second [strict]
foldll [?, ?, strict]
mapAccuml [?, ?, strict]
nonnull [?, strict]
lexDigits [strict]
asciiTab []
match [strict, ?]
isOctDigit [strict]
isHexDigit [?]
I2937_lexLitChar [strict]
I2934_lexEsc [?, strict]
I2928_lexEsc [strict]
I2931_lexEsc [strict]
lexEsc [strict]
lexLitChar [strict]
```

```
isSingle [?]
isSym [?]
isSym1 [?]
I2913_lex [?, ?, strict]
I2922_lexExp [?, strict]
I2925_lexExp [?, ?, ?, strict]
lexExp [strict]
I2918_lexFracExp [?, ?, strict]
I2916_lexFracExp [strict]
lexFracExp [strict]
isIdChar [strict]
Lam3717_lex [strict]
I2910_lex [strict]
lexStrItem [strict]
I2906_lexString [?, ?, strict]
I2904_lexString [strict]
lexString [strict]
I2901_lex [strict]
Lam3716_lex [strict]
lexNest [?, strict]
lex [strict]
rLookup [strict, ?]
oLookup [strict, ?]
rUpdate [strict, ?, ?]
oUpdate [strict, ?, ?]
bUpdate [strict, ?, ?]
bAlloc [strict, ?]
allocateSc [strict, strict]
I3270_argOffset [?, strict]
argOffset [?, strict]
LmakeSpine [?]
makeSpine [strict]
cons12 [?, ?]
le [strict, strict]
ge [strict, strict]
enumFromThen [strict, strict]
enumFromThenTo [strict, strict, ?]
compileArgs [strict, ?]
I3276_compileAlts [?, ?, strict]
compileAlts [?, strict, ?]
saturatedCons [strict]
I3273_compileE [strict]
compileLetrec´ [strict, ?]
compileLetrec [?, ?, ?, ?]
compileLet´ [strict, ?]
compileLet [?, strict, ?, ?]
I3279_compileC [strict]
```

```
compileC´ [?, ?, ?]
LcompileCS [?, strict, ?]
compileCS [strict, ?]
compileC [strict, ?]
LcompileE [?, ?]
compileE [strict, ?]
compileE´ [?, ?, ?]
compileR [strict, ?]
compileSc [strict]
getArg [strict]
initialCode []
+ [strict, strict]
- [strict, strict]
* [strict, strict]
/ [strict, strict]
_+ [strict, strict]
_- [strict, strict]
_* [strict, strict]
_/ [strict, strict]
_negate [?]
negate [strict]
== [strict, strict]
~= [strict, strict]
< [strict, strict]
<= [strict, strict]
> [strict, strict]
>= [strict, strict]
_== [strict, strict]
_~= [strict, strict]
_< [strict, strict]
_<= [strict, strict]
_> [strict, strict]
_>= [strict, strict]
_ord [?]
_chr [?]
ord [strict]
chr [strict]
strict [strict, strict]
36.56user 1.44system 0:39.79elapsed 95%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+0minor)pagefaults 0swaps
```

# Literaturverzeichnis


[AJ89a] Lennart Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *Computer Journal*, 32(2), 1989.

[AJ89b] Lennart Augustsson and T. Johnsson. Parallel graph reduction with the $\langle \nu, G \rangle$ -machine. In *Conference on Functional Programming and Computer Architecture*, Imperial College, London, September 1989. ACM Press.

[Aug84] Lennart Augustsson. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pages 218–27, August 1984.

[Aug94] Lennart Augustsson. Private Korrespondenz, 1994.

[Bab87] Robert Laurence Baber. *The Spine of Software*. John Wiley & Sons, Chichester, 1987.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:613–??, 1978.

[Bur91] Geoffrey L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, London, 1991.

[Dav92] A. J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, Cambridge, 1992.

[Dij76] Edsger W. Dijkstra. *A Dicipline of Programming*. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[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.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction, Montreal*, pages 58–69, June 1984.

[Jon93] Mark P. Jones. *Gofer*, Februar 1993.

[Les88] David R. Lester. Combinator graph reduction: A congruence and its applications. Technical report, Oxford University, 1988.

[Nö92] Eric Nöcker. Strictness analysis by abstract reduction in orthogonal term rewriting systems. Technical report, University of Nijmegen, Department of Computer Science, 1992.

[Nö93] Eric Nöcker. Strictness analysis using abstract reduction. Technical report, University of Nijmegen, Department of Computer Science, 1993.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, London, 1987.

[PJ93] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. Version 2.5. Technical report, University of Glasgow, Department of Computing Science, 1993.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: A Tutorial*. Prentice-Hall International, London, 1991.

[PJP] Simon L. Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. Draft.

# Index

# Programm Index

In diesem Index wird auf die Definitionen der Funktionen gezeigt (unterstrichene Einträge) und auf die Verweise auf Programmtext aus dem laufenden Text (nicht unterstrichende Einträge).