Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Biologie und Informatik
Institut für Informatik

Diplomarbeit

Implementierung eines generischen Interpreters für Termersetzungssysteme höherer Ordnung auf Basis einer strukturellen operationalen Semantik

Christopher Stamm

15. März 2004

eingereicht bei

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz / Softwaretechnologie

Danksagung

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

Christopher Stamm

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 15. März 2004

Christopher Stamm

Inhaltsverzeichnis

1 Einführung

1.1 Motivation und Ziele

Die funktionale Programmierung hat in der Informatik eine große Bedeutung, sowohl in der Theorie als auch in der Praxis. Aufgrund des deklarativen Charakters der funktionalen Programmiersprachen bestehen die Programme nicht, wie bei imperativen Sprachen, aus einer Sequenz von Anweisungen, sondern es handelt sich eher um eine formale Spezifikation des Ergebnisses. Dadurch sind die Programme besser lesbar, schneller zu verstehen und somit auch leichter zu warten. Ebenso lassen sich funktionale Programme schneller und leichter schreiben, da man auf intuitive Weise das Programm strukturieren und in Teilprobleme aufteilen kann, bei immer komplexer werdenden Anforderungen an die Softwaresysteme also ein enormer Vorteil, zumal funktionale Programme auch oftmals deutlich kürzer ausfallen als entsprechende imperative Programme.

Zudem haben funktionale Programme eine einfachere Semantik als imperative, denn diese ist unabhängig von einem zugrunde liegenden Maschinenmodell und benötigt keinen globalen Zustand. Dadurch ist es vergleichsweise leicht, logische Kalküle aufzustellen, die aussagekräftige Analysen der Reduktionsmechanismen erlauben. Von besonderem Interesse sind dabei Aussagen über die Gleichheit von Programm-Fragmenten, denn diese sind die Grundlage für Programm-Transformationen, mit denen sich z.B. die Anzahl der benötigten Reduktionsschritte deutlich verringern lässt. Der übliche Gleichheits-Begriff ist hier der Begriff der *kontextuellen Gleichheit*, bei der zwei Ausdrücke dann gleich sind, wenn sie an beliebigen Stellen des Programms vertauscht werden können, ohne dass ein anderes Verhalten festgestellt werden kann (in der Regel betrachtet man die Terminierung).

Allerdings ist diese kontextuelle Gleichheit schwierig zu handhaben, da sie kein schrittweises Vorgehen zulässt. Daher verwendet man lieber eine Relation, die ein Gedankenexperiment beschreibt, bei der die Ausdrücke nach und nach mit neuen Argumenten versorgt werden und entsprechend das Terminierungsverhalten beobachtet wird. Dazu muss man jedoch erst zeigen, dass diese sog. *Bisimulation* mit der kontextuellen Gleichheit übereinstimmt. Dies kann für viele Kalküle gezeigt werden, aber diese zentrale Aussage kann auch zwischen sich ähnelnden Kalkülen nicht ohne weiteres übertragen werden.

In dieser Stelle erweist es sich als nützlich, Reduktionsmechanismen auf abstrahiertere Art und Weise mittels Termersetzungssystemen zu betrachten. Diese bestehen aus einem einfachen Regelsystem zur Ersetzung von Termen einer Sprache. *Termersetzungssysteme höherer Ordnung* erlauben die Betrachtung von Funktionseinsetzungen in λ-Termen, inklusive Variablenbindung. Somit kann auch der der funktionalen Programmierung zu Grunde liegende λ-Kalkül mittels eines Termersetzungssystems dargestellt werden.

Besonders nützlich sind die Termersetzungssysteme, da bei bestimmten Einschränkungen des Regelsystems sichergestellt werden kann, dass die Bisimulations-Relation eine *Kongruenz* ist, d.h. eine Äquivalenzrelation, die die Programmstruktur respektiert. Von dieser Eigenschaft ausgehend kann man dann relativ leicht zeigen, dass die Bisimulation

mit der kontextuellen Gleichheit übereinstimmt, falls einige zusätzliche Eigenschaften erfüllt sind.

Die Systeme, für die diese Eigenschaft gilt, sind nun diejenigen, die sich an die *strukturelle operationale Semantik* der strukturierten Auswertungswertungssysteme nach [How96] bzw. des GDSOS[1]-Formats nach [San97] halten. Eine strukturelle operationale Semantik ist dabei die Beschreibung der Ausführung einer Operation bzw. einer Reduktion über die Struktur der Ausdrücke. Das GDSOS-Format kann man direkt als Beschreibung eines Termersetzungssystems deuten; die damit beschriebene Auswertungsrelation hält sich an die Regeln der strukturierten Auswertungssysteme (Beweis im Rahmen dieser Arbeit), weshalb wir in diesem Zusammenhang entsprechende Termersetzungssysteme auch als strukturierte Reduktionssysteme bezeichnen wollen.

Ziel dieser Arbeit ist zum einen, verschiedene Kernsprachen und Kalküle auf ihre Tauglichkeit zur Integration in ein solches strukturiertes Reduktionssystem zu überprüfen, zum anderen die Implementierung eines solchen Systems, das eine strukturelle operationale Semantik eines Termersetzungssystems einliest und gemäß dieser Ausdrücke auswerten kann. Dieses System wird die Bedingungen der strukturierten Auswertungs- bzw. Reduktionssysteme überprüfen, jedoch nicht immer auf diesen Beschränkungen bestehen. Stattdessen werden wir noch einige Erweiterungen implementieren, um eine möglichst große Anzahl verschiedener Kalküle zu unterstützen. So wollen wir neben dem λ-Kalkül und Kalkülen, die im Wesentlichen Erweiterungen des λ-Kalküls darstellen, auch einen let-Kalkül auswerten können, welcher Sharing explizit kodiert. Unser System wird dabei nicht nur mit zahlreichen Schnittstellen für automatische Beweissysteme, die auf Reduktion bzw. Ersetzung basieren, versehen sein, sondern auch über einen Interpreter verfügen, der einen gegebenen Ausdruck vollständig auswertet, unter Angabe aller Zwischenschritte. Somit könnte man das System auch als Programmiersprache für abstrakte Kalküle benutzen.

1.2 Überblick

Zunächst wollen wir uns in Kapitel 2 die besonderen Eigenschaften funktionaler Programmiersprachen genauer anschauen und wichtige Begriffe, wie z.B. Striktheit, eingeführen. Wir gehen auch noch einmal auf die Unterschiede zu imperativen Sprachen ein und verdeutlichen dabei insbesondere, welche Vorteile es hat, funktionale Programmierung zu verwenden, wobei natürlich auch die wenigen Nachteile nicht verschwiegen werden. Als Vertreter für eine funktionale Programmiersprache stellen wir anschließend die Sprache Haskell vor, die wir dann auch in Kapitel 6 für die Implementierung unseres Softwaresystems nutzen wollen.

In Kapitel 3 setzen wir uns dann mit dem Lambda-Kalkül auseinander, welcher die stark vereinfachte Grundlage der funktionalen Programmierung darstellt. Zunächst begutachten wir die Konversionsregeln des (klassischen) Lambda-Kalküls nach [Bar84]. Anschließend führen wir im Rahmen einer näheren Betrachtung des „lazy“ Lambda-Kalküls von Abramsky [Abr90] die Begriffe der kontextuellen Gleichheit und der Bisimulation formal

[1] Globally Deterministic Structural Operational Semantics

ein. Am Ende des Kapitels wird auch noch kurz der Call-by-Value Lambda-Kalkül erläutert.

Anschließend folgt in Kapitel 4 eine Vorstellung weiterer Kernsprachen und Kalküle. Neben den Kernsprachen KFP, KFP+ und PCF, bei denen es sich im Wesentlichen um eine Erweiterung des Lambda-Kalküls um Konstruktoren und Fallunterscheidungen handelt, betrachten wir auch den Call-by-Need Lambda-Kalkül, dessen Reduktionsregeln sich stark vom üblichen Lambda-Kalkül unterscheiden.

Nachdem wir uns somit in die Grundlagen der funktionalen Programmierung und der zugrunde liegenden Kalküle eingearbeitet haben, folgt in Kapitel 5 die Betrachtung der Termersetzungssysteme höherer Ordnung und wie wir die vorher betrachteten Kalküle mit ihrer Hilfe darstellen können. Außerdem untersuchen wir die speziellen Regelformate der GDSOS-Reduktionssysteme [San97] und der strukturierten Auswertungssysteme nach Howe [How96]. Diese haben u.a. die besondere Eigenschaft, dass gewährleistet ist, dass Bisimulation eine Kongruenz ist.

Kapitel 6 ist schließlich der Implementierung unseres Softwaresystems gewidmet. Zunächst werden einige Vorüberlegungen über die Funktionalität der Software angestellt, da wir das Konzept der Termersetzungssysteme höherer Ordnung noch so erweitern wollen, dass wir auch den Call-by-Need Lambda-Kalkül darstellen können. Dazu benötigen wir insbesondere spezielle Verarbeitungsmöglichkeiten für Kontexte. Anschließend werden die implementierten Programmteile ausführlich erläutert. Zum Abschluss des Kapitels begutachten wir die mit unserem Softwaresystem erzielten Ergebnisse.

In Kapitel 7 folgt eine kurze Zusammenfassung und ein Ausblick auf künftige Aufgaben.

2 Funktionale Programmierung

Im Folgenden soll ein Überblick über die Besonderheiten der funktionalen Programmierung gegeben werden und dabei, neben Betrachtungen der unterschiedlichen Konzepte verschiedener funktionaler Programmiersprachen, auch eine Erläuterung der grundlegenden Begriffe erfolgen. Da wir in der Einleitung bereits einige Vorteile funktionaler Programmiersprachen gegenüber imperativen Vertretern gesehen haben, wollen wir in diesem Kapitel auch noch einmal einen detaillierteren Vergleich anstellen.

Die Beispiel-Programme in Abschnitt 2.1 können in jeder funktionalen Programmiersprache implementiert werden, wobei wir uns jedoch syntaktisch an die Programmiersprache Haskell halten. Diese wird in Abschnitt 2.3 näher vorgestellt. Ausführlichere Betrachtungen der hier vorgestellten Konzepte und weiterführende Themen findet man in [Sch00] und [Bir98].

2.1 Das funktionale Konzept

Funktionale Programmiersprachen gehören zu den deklarativen Sprachen und unterscheiden sich somit stark von imperativen (und objektorientierten) Sprachen: Während sich imperative Programmiersprachen an der von-Neumann-Rechnerarchitektur mit Prozessor und Arbeitsspeicher orientieren, sind deklarative Sprachen an mathematischen Konzepten und Notationen angelehnt. Funktionale Programmiersprachen richten sich nach dem Konzept mathematischer Funktionen, also der Abbildung von Eingabewerten auf Ausgabewerte.

In der funktionalen Programmierung besteht ein Programm somit im Wesentlichen aus Funktionsdefinitionen, Funktionsanwendungen und Funktionskompositionen. Die Anwendung einer Funktion auf ihre Argumente ist dabei das zentrale Element, da die Auswertung von Ausdrücken im Vordergrund steht und nicht, wie bei imperativen Sprachen, die Ausführung von Befehlen: Die Ausführung eines Programms besteht nämlich ausschließlich aus der Ausführung/Reduktion von Funktionsanwendungen[2]. Die Anwendung einer Funktion auf ihre Argumente wird einfach durch Einsetzen der Argumente in die Funktionsdefinition bzw. in den Funktionsrumpf reduziert; der Ausdruck wird dabei durch sein Reduktionsergebnis ersetzt, welches dann gegebenenfalls weiter reduziert wird, bis ein *Wert* erreicht wird. Ein Wert ist dabei Teil einer Menge von Ausdrücken, die wir als Ergebnis einer Auswertung akzeptieren. Somit ist das einzige Resultat eines Ausdruckes bzw. Programms der Wert des Ausdruckes.

Beispiel 2.1.1.
Eine Funktion, die das Quadrat ihres Eingabewertes zurückliefert, kann man wie folgt definieren:

```
quadrat x = x * x
```

[2] Wir betrachten Fallunterscheidungen hier auch als spezielle Funktionen.

Ein einfaches Programm wäre nun:

```
main = quadrat 3
```

Dieses Programm macht nichts weiter, als das Quadrat von 3 zu berechnen. Die Auswertung erfolgt durch Einsetzen der Argumente in den Funktionsrumpf (die rechte Seite der Funktionsdefinition) und Vereinfachung der entstehenden Ausdrücke:

```
main → quadrat 3 → 3 * 3 → 9
```

Funktionen können selbstverständlich rekursiv definiert sein, also sich selbst aufrufen. In der Regel sind Funktionen aber auch als Argumente und Resultate von Funktionen zugelassen, dies wird allgemein als *höhere Ordnung*[3] bezeichnet. Zudem können auch ungesättigte Funktionsanwendungen wieder als eigenständige Funktionen benutzt werden, d.h. man kann eine Funktion auf weniger Argumente als in der Funktionsdefinition vorgesehen anwenden und erhält so eine neue Funktion. So kann z.B. `(+1)` als Funktion, die ihr Argument um eins erhöht, verwendet werden. Auf diese Weise ist es auch möglich, alle Funktionen mit mehreren Argumenten auf Funktionen mit nur einem Argument zurückzuführen, so dass man statt einer Funktionsanwendung auf mehrere Argumente einfach mehrere Funktionsanwendungen auf ein Argument auswerten kann. Dieses Verfahren wird *Currying*[4] genannt.

Beispiel 2.1.2.
`(3 * 3)` kann man auch als `((*) 3) 3` betrachten.

2.1.1 Referentielle Transparenz

Im Gegensatz zur imperativen Programmierweise gibt es in der funktionalen Programmierung keine Variablenzuweisung. Variablen sind ausschließlich Eingabewerte einer Funktion und werden nicht mehr geändert. Dies stellt sicher, dass Berechnungen nicht durch *Seiteneffekte* gestört werden: Stützen nämlich Funktionen in imperativen Sprachen ihre Berechnungen auf globale Variablen, so kann der Inhalt dieser Variablen an beliebigen Stellen im Programm geändert werden – das Funktionsergebnis hängt somit wesentlich vom globalen (Speicher-)Zustand und nicht mehr nur von den Argumenten der Funktion ab. Zwei gleiche Funktionsaufrufe (mit gleichen Argumenten) an verschiedenen Stellen des Programms können also in Seiteneffekt-behafteten Programmiersprachen zu unterschiedlichen Ergebnissen führen. Dies widerspricht der mathematischen Sicht, dass Funktionen wohldefiniert - also eindeutig - sein müssen. Funktionale Programmiersprachen, die keinerlei Seiteneffekte erlauben, nennt man *pur*. In puren Sprachen gilt das Prinzip der *referentiellen Transparenz:* Der Wert eines Ausdrucks hängt einzig und allein von seinen Teilausdrücken ab. Somit ist gewährleistet, dass gleiche (Teil-)Ausdrücke an verschiedenen Stellen des Programms immer den gleichen Wert haben. Dies ermöglicht, Ausdrücke in beliebiger Reihenfolge auszuwerten, sofern die Auswertung terminiert, mehr dazu in Abschnitt 2.1.3. In diesem Fall ist also auch eine parallele Auswertung möglich.

---

[3] engl. *higher order*
[4] Nach dem Logiker Haskell B. Curry

Für Programmanalysen ist referentielle Transparenz ebenfalls von enormer Bedeutung, denn diese garantiert *Ersetzbarkeit:* Jeder Ausdruck kann durch einen beliebigen Ausdruck mit gleichem Wert ersetzt werden. Dies vereinfacht z.B. Gleichheitsanalysen und bietet weitreichende Möglichkeiten zur Optimierung, da man komplizierte Ausdrücke durch einfachere ersetzen kann, wenn die Gleichheit bewiesen ist. Gilt referentielle Transparenz hingegen nicht, so ist es relativ schwierig, Aussagen über Programmfragmente zu treffen, da man stets den Maschinenzustand mitbetrachten muss.

2.1.2 Bedeutung eines Ausdrucks

Nicht bei allen Ausdrücken ist eine Auswertung möglich, da es Funktionen gibt, die nicht terminieren. Das einfachste Beispiel zeigt folgende Funktionsdefinition:

```
bot = bot
```

Welche Bedeutung bzw. welchen Wert hat ein solcher nichtterminierender Ausdruck? Um sicherzustellen, dass alle Ausdrücke, auch die nicht auswertbaren bzw. nicht-terminierenden, eine Bedeutung haben, ordnen wir jedem Ausdruck einen semantischen Wert zu. Üblicherweise wird nichtterminierenden Ausdrücken ein formaler Wert $\perp$ zugewiesen und alle nichtterminierenden Ausdrücke werden als gleich betrachtet. Jeder Ausdruck *bezeichnet* nun einen formalen Wert, man spricht von einer *denotationalen*[5] *Semantik*. Im Gegensatz dazu betrachtet man bei der *operationalen Semantik* die Ausführung einer Operation, d.h. man beschreibt die Reduktion. Die denotationale Semantik beschreibt also allein das Ergebnis, die operationale Semantik hingegen die Berechnung. Die Übereinstimmung zwischen denotationaler und operationaler Semantik wird *Adäquatheit* der Auswertung[6] genannt: Ein Ausdruck behält dabei seinen semantischen Wert auch nach seiner Reduktion bei, die Bedeutung verändert sich während der Auswertung also nicht.

2.1.3 Auswertungsstrategien

Welche Rolle spielt nun die Reihenfolge, in der (Unter-)Ausdrücke ausgewertet werden? Bei Funktionsanwendungen haben wir die Wahl, ob wir zuerst die Argumente auswerten (*strikte Auswertung*) oder die noch nicht ausgewerteten Ausdrücke in die Funktionsdefinition einsetzen (*nicht-strikte Auswertung*).

Beispiel 2.1.3. Betrachten wir folgenden Ausdruck:

```
quadrat (2 + 1)
```

Strikte Auswertung:

```
quadrat (2 + 1) → quadrat 3 → 3 * 3 → 9
```

Bei nicht-strikter Auswertung ergeben sich zwei Möglichkeiten:

---

[5] engl. *to denote* – bezeichnen, bedeuten
[6] engl. *computational adequacy*

```
quadrat (2 + 1) → (2 + 1) * (2 + 1) →
  3 * (2 + 1) → 3 * 3 → 9

quadrat (2 + 1) → (2 + 1) * (2 + 1) →
  (2 + 1) * 3 → 3 * 3 → 9
```

Wie wir sehen, führen alle drei Auswertungsstrategien zum selben Wert. Es gilt allgemein nach dem ersten Satz von Church-Rosser: Wenn zwei verschiedene Reduktionsfolgen terminieren, haben beide den selben Wert (vgl. [Sch97]).

Reduktionsfolgen mit strikter Auswertung sind in *applikativer Reihenfolge*. Die Auswertungsstrategie heißt dementsprechend *Applikations-* oder *Anwendungsordung*. Eine andere gebräuchliche Bezeichnung ist *Call-by-Value*.

Bei nicht-strikter Auswertung ist es üblich, den am weitesten links (und am weitesten oben) stehenden Ausdruck zuerst auszuwerten. Man spricht von *normaler Reihenfolge.* Die Auswertungsstrategie heißt *Normalordnung* oder auch *Call-by-Name*.

Wie in Beispiel 2.1.3. zu sehen ist, braucht die normale Reihenfolge unter Umständen mehr Reduktionsschritte als die applikative Reihenfolge. Dies ist aber auch umgekehrt der Fall, da in Anwendungsordnung möglicherweise Argumente ausgewertet werden, die gar nicht für die Berechnung gebraucht werden. Als Beispiel betrachten wir die Funktion `k`, die zwei Argumente hat und einfach ihr erstes Argument zurückliefert:

```
k x y = x
```

Offensichtlich kann ein Ausdruck `k v t`, bei dem `v` ein Wert und `t` ein schwierig zu berechnender Ausdruck ist, in Normalordnung mit einem Reduktionsschritt ausgewertet werden, während in Anwendungsordnung sehr viele Schritte notwendig sind. Im Extremfall terminiert `t` nicht, dies führt dazu, dass auch `k v t` in Anwendungsordnung nicht ausgewertet werden kann, in Normalordnung aber sehr wohl. Allgemein gilt nach dem zweiten Satz von Church-Rosser: Falls eine beliebige Auswertung terminiert, dann tut dies auch eine Auswertung in Normalordnung (vgl. [Sch97]).

Den Nachteil der Normalordnung, dass Ausdrücke bei Funktionseinsetzungen möglicherweise kopiert werden und so mehrfach berechnet werden müssen, kann man durch *Sharing* beheben, d.h. statt Kopien anzulegen, werden Referenzen auf einen einzigen Ausdruck angelegt, der von allen referenzierenden Knoten geteilt wird. Man wechselt also in der Implementierung von einer Baum- zu einer Graphstruktur. In unserer schriftlichen Reduktion entspricht dies einer parallelen Auswertung der kopierten Ausdrücke:

```
quadrat (2 + 1) → (2 + 1) * (2 + 1) → 3 * 3 → 9
```

Jetzt werden nur noch so viele Reduktionsschritte gebraucht wie in Anwendungsordnung. Allgemein gilt: Normalordnung mit Sharing führt zu einer optimalen Anzahl an Reduktionsschritten (vgl. [Sch97]). Diese Auswertungswertungsstrategie wird

üblicherweise als *lazy*[7] oder *Call-by-Need* bezeichnet. Bei Vergleichen zur lazy-Auswertungsstrategie spricht man bei der Anwendungsordnung auch gerne von *eager evaluation*: statt faul fleißig, da jeder Ausdruck ausgewertet wird.

2.1.4 Striktheit

Üblicherweise werden funktionale Programmiersprachen in strikte und nicht-strikte Sprachen unterteilt, wobei man man hier mit strikt die Auswertung in Anwendungsordnung und mit nicht-strikt die lazy-Auswertung meint. Vertreter der strikten Sprachen sind Scheme ([Dyb96]) und ML ([MTH+97]). Nicht-strikt sind z.B. Miranda ([Tur85]), Clean ([PvE93]) und Haskell ([PJ03]).

Fehlende Striktheit hat bedeutende Konsequenzen: Sie ermöglicht nämlich unendliche (rekursiv definierte) Datenstrukturen. So kann beispielsweise in Haskell eine Funktion definiert werden, die eine unendlich lange Liste mit Einsen zurückliefert:

```
einser = 1:einser
```

Der Ausdruck `einser` wird nun ausgewertet zu `1:1:1:1`... und terminiert somit nicht. Allerdings kann dieser Ausdruck einer Funktion übergeben werden, die nur eine bestimmte Anzahl von Listenelementen untersucht und somit terminiert. Aufgrund der nicht-strikten Auswertung wird der nächste Rekursionsschritt von `einser` nämlich immer erst dann vorgenommen, wenn ein weiteres Listenelement ausgewertet werden soll. Unendliche Datenstrukturen bieten somit neue sinnvolle Ausdrucksmöglichkeiten, man denke z.B. statt der einfachen Liste sich wiederholender Einsen an die Liste aller Primzahlen.

In der Regel gibt es auch in nicht-strikten Sprachen spezielle Funktionen, die verlangen, dass ihre Argumente ausgewertet vorliegen – üblicherweise die (internen) arithmetischen Funktionen z.B. + und *. Zudem bezeichnet man Funktionsargumente, die im Laufe einer (lazy-)Auswertung auf jeden Fall ausgewertet werden, als strikte Argumente, d.h. falls für dieses Argument ein nichtterminierender Ausdruck eingesetzt wird, so terminiert die Auswertung der Funktionsanwendung nicht.

Definition 2.1.4. Eine einstellige Funktion f ist *strikt in ihrem Argument*, gdw. $f \perp = \perp$.

Definition 2.1.5. In einer n-stelligen Funktion f ist das i-te Argument strikt, gdw. für beliebige Ausdrücke $x_1, ..., x_{i-1}, x_{i+1}, ..., x_n$ gilt: $f\, x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n = \perp$.

2.1.5 Typsystem

Viele funktionale Programmiersprachen verfügen über ein strenges, statisches Typsystem. Typfehler werden somit bereits während des Kompilierens erkannt. Im Gegensatz zu einem dynamischen Typsystem, wie es viele imperativen Sprachen besitzen, sind somit zur Laufzeit keine Typfehler mehr möglich. Unterstützt das Typsystem zudem Typvariablen, so ist Polymorphismus möglich, die gleiche Funktion kann somit für alle Datentypen

[7] engl. faul. Im Deutschen wird teilweise auch der Begriff „verzögerte“ Auswertung benutzt.

eingesetzt werden, die von den innerhalb der Funktionsdefinition verwendeten Zugriffsfunktionen unterstützt werden.

2.2 Vor- und Nachteile funktionaler Programmierung

Wir haben bereits einige Vorteile der funktionalen Programmierung gesehen, ebenso entscheidende Unterschiede zur imperativen Programmierweise. Prinzipiell sind beide Programmier-Methoden gleich mächtig, denn die zu Grunde liegenden formalen Modelle – bei der funktionalen Programmierung der (im nächsten Kapitel vorgestellte) Lambda-Kalkül und bei der imperativen Programmierung die Turing-Maschine – sind äquivalent, in dem Sinne, das alles, was das eine Modell berechnen kann, auch im anderen Modell berechnet werden kann. Gemäß der Church-Turing-These wird vermutet, dass überhaupt alles, was berechnet werden kann, in diesen beiden Modellen berechnet werden kann.[8] Entscheidende Vergleichs-Kriterien sind deshalb folgende: Ressourcen-Bedarf zur Ausführung eines Programms und der Arbeitsaufwand zur Erstellung (und Wartung) eines Programms.

Wir wollen nun an dieser Stelle ein Beispiel aus [PJ98] betrachten, das eigentlich alle wichtigen Unterschiede aufzeigt: den wohlbekannten Quicksort-Algorithmus. Wir vergleichen dabei eine Implementierung in der funktionalen Programmiersprache Haskell mit einer Implementierung in der imperativen Sprache C. Dabei interessieren wir uns nicht für Implementierungsdetails, sondern vergleichen nur die grundlegenden Konzepte. Einige verwendete Techniken des Haskell-Programms werden in Abschnitt 2.2 erklärt.

Beispiel 2.2.1.

Quicksort in Haskell:

```
qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
                 where
                   elts_lt_x   = [y | y <- xs, y < x]
                   elts_greq_x = [y | y <- xs, y >= x]
```

Quicksort in C:

```
qsort( a, lo, hi ) int a[], hi, lo;
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
```

[8] Dabei wird freilich von einem intuitiven Berechnungsbegriff ausgegangen, so dass diese These nicht bewiesen werden kann. Dennoch ist sie allgemein akzeptiert.

```
    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    t = a[l];
    a[l] = a[hi];
    a[hi] = t;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

Als erstes fällt auf: Das Haskell-Programm ist deutlich kürzer. Dieses Beispiel ist sicherlich ein Extremfall, dennoch zeigt die Erfahrung, dass funktionale Programme allgemein kürzer sind ([PJ98] spricht von zwei- bis zehnmal kürzer). Dies liegt vor allem an den häufigen Variablenzuweisungen und den Steuerungsbefehlen der Schleifen im C-Programm. Diese machen imperative Programme auch oft unübersichtlich, denn die Elemente der Spezifikation werden mit der Steuerung der Rechnerarchitektur vermengt. In der funktionalen Programmierung entfallen diese Elemente, da es keine Variablenzuweisungen und Schleifen gibt, statt Schleifen wird im wesentlichen rekursives Programmieren verwendet. Zudem wird in vielen funktionalen Programmiersprachen der Speicher automatisch verwaltet, während es in imperativen Sprachen nicht unüblich ist, dass der Programmierer dafür Sorge tragen muss, dass sein Programm an den richtigen Stellen Speicher alloziert und wieder freigibt.

Auf der anderen Seite gibt die explizite, maschinennahe Steuerung der Rechnerarchitektur dem gewieften Programmierer die Möglichkeit, die Effienz eines Programms (in Bezug auf den Ressourcenverbrauch) erheblich zu verbessern. Das obige C-Quicksort-Programm sortiert die Eingabe an Ort und Stelle, d.h. ohne zusätzlichen Speicherverbrauch und hat somit nicht nur einen deutlich niedrigeren Platzbedarf als das Haskell-Programm, sondern läuft auch wesentlich schneller, da die zusätzliche Speicherverwaltung im Haskell-Programm viel Zeit in Anspruch nimmt. Das C-Programm hat also die Speicherorganisation während der Laufzeit, unter Inkaufnahme einer höheren Komplexität des Algorithmus, minimiert.

Fazit: Imperative Programme sind oft schneller, funktionale dafür einfacher. Mit imperativer Programmierung kann man den Ressourcenverbrauch einer Software verringern, mit funktionaler Programmierung hingegen den Arbeitsaufwand zur Erstellung und Wartung eines Programms. Wenn man bedenkt, dass Softwaresysteme immer komplexer werden und somit die Kosten für Entwicklung und Wartung immer mehr

steigen, gleichzeitig aber moderne Computer immer leistungsfähiger werden, kann man annehmen, dass die Vorteile der funktionalen Programmierung in Zukunft immer mehr an Bedeutung gewinnen werden. Schließlich lassen sich die Vorteile der beiden Programmierweisen auch kombinieren: Moderne funktionale Sprachen bieten ausreichend Möglichkeiten zum Im- und Export von Modulen bzw. Bibliotheken in standardisierten Formaten (z.B. DLL[9]).

In Bezug auf Modularisierung bieten gerade nichtstrikte Programmiersprachen auch neue Möglichkeiten (siehe [Hug89/90]), denn diese können z.B. Module mit Definitionen unendlicher Datenstrukturen verarbeiten und erlauben eine starke Bindung, indem nur die Teile eines Unterprogramms ausgewertet werden, die auch zum Erreichen des Ergebnisses nötig sind.

2.2.1 Interaktion/IO

Ein wichtiger Aspekt purer funktionaler Sprachen wurde bisher allerdings noch unterschlagen: Es gibt Situationen, in denen Seiteneffekte durchaus erwünscht sind. Jegliche Art von Interaktion mit dem Benutzer bzw. jede IO[10]-Aktion stellt nämlich einen Seiteneffekt dar. Imperative Sprachen sind dabei natürlich ganz in ihrem Element. Wie sieht es bei puren funktionalen Sprachen aus?

Für funktionales IO gibt es unterschiedliche Ansätze, und dieses Thema ist auch nach wie vor aktuelles Forschungsgebiet. Eine gängige Methode ist monadisches IO, das in Abschnitt 2.3.7 für Haskell erklärt wird. Die zugrundeliegende Idee ist dabei, den globalen Zustand als explizites Argument anzugeben (so dass die referentielle Transparenz gewahrt bleibt) und eine Sequentialisierung zu erzwingen, d.h. alle Funktionen, die auf den globalen Zustand zugreifen, müssen in ihrer vorgegebenen Reihenfolge abgearbeitet werden. Diese Methode führt also zu einer Art imperativem Programmierstil innerhalb des funktionalen Programms.

2.3 Haskell

Haskell ist eine pure, nicht-strikte funktionale Programmiersprache mit statischem, polymorphem Typsystem. Die wichtigsten Implementierungen sind der *Glasgow Haskell Compiler* (mit Interpreter *GHCi*) und der Interpreter *Hugs*. Beide sind im Internet[11] frei verfügbar. Wir wollen nun einige interessante Aspekte des Sprachumfangs beleuchten, die wir auch in unseren Programmen verwenden wollen.

2.3.1 Module

Haskell erlaubt die Gliederung eines Programms in mehrere Module, die jeweils innerhalb einer eigenen Datei definiert sind. Eine besondere Bedeutung hat dabei das Modul `Main`,

[9] Dynamic Link Library
[10] Input-Output
[11] www.haskell.org

dass eine Funktion `main` enthalten (und exportieren) muss, denn diese wird beim Start des Programms ausgeführt bzw. ausgewertet.

Eine Moduldefinition wird eingeleitet durch das Schlüsselwort `module`, gefolgt von einer Liste der exportierten Bezeichner und einer Liste von `import`-Anweisungen zum Import von Modulen. Beispiel:

```
module Main(main) where

  import Berechnungen
  …
  main = …
  …
```

Module sind idealerweise so gestaltet, dass sie auch von anderen Programmen wiederverwendet werden können.

2.3.2 Datentypen

In Haskell können (algebraische) Datentypen durch Angabe ihrer Konstruktoren definiert werden. Der Typ Bool ist zum Beispiel wie folgt definiert:

```
data Bool = True | False
```

Konstruktoren können auch mehrstellig sein, d.h. auf weitere Typen zurückgreifen. Da es sich um ein polymorphes Typsystem handelt, können auch Typvariablen verwendet werden. So ist z.B. der wichtige Listen-Typ wie folgt definiert:

```
data [a] = [] | a : [a]
```
[12]

Die Datenkonstruktoren `:` und `[]` werden als Cons und Nil bezeichnet, bei `a` handelt es sich um eine Typvariable – somit können Listen beliebige Typen enthalten. Da Listen eine besondere Rolle spielen, ist auch eine besondere Syntax für sie vorgesehen: So kann man statt `1:2:[]` auch `[1,2]` schreiben.

Es ist auch möglich, Typsynonyme zu definieren. So wird z.B. der Typ String in Haskell einfach auf eine Liste zurückgeführt:

```
type String = [Char]
```

Selbstverständlich steht aber auch für Strings in Haskell die gewohnte Syntax zur Verfügung, man kann anstatt `['A', ' ', 's', 't', 'r', 'i', 'n', 'g']` ganz normal `"A string"` schreiben.

[12] Genau genommen sind Listen intern definiert, dies ist nur eine anschauliche Definition, die der internen entspricht – leider ist diese nicht ganz Haskell-konform (Kommentar aus [PJ03]: Not legal Haskell; for illustration only).

2.3.3 Pattern Matching

Ein Hilfsmittel, um die Konstruktoren eines Datentyps zu unterscheiden und gegebenenfalls auf ihre Inhalte zuzugreifen, ist Pattern Matching. So ist z.B. die Funktion `map`, die eine beliebige Funktion auf alle Elemente einer Liste anwendet, mittels Pattern Matching definiert:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Wie man sieht, wird die Funktion einfach für jeden Konstruktor separat definiert. Die Muster bzw. Pattern müssen aber nicht aus einfachen Konstruktoren bestehen, sondern können auch geschachtelt sein. Bei einem Funktionsaufruf wird nun überprüft, mit welchem Muster ein übergebenes Argument übereinstimmt („Match“) und die entsprechende Funktionsdefinition für die Reduktion verwendet. Die Überprüfung findet dabei sequentiell von oben nach unten statt, wobei einfach die erste gefundene Übereinstimmung verwendet wird. Ein besonderes Pattern ist `_` („Wildcard“), dieses teilt dem Compiler mit, dass das entsprechende Argument uninteressant ist und führt immer zu einem Match. Pattern Matching ist auch innerhalb eines `Case`-Ausdrucks möglich, die Funktion `map` könnte man also auch wie folgt definieren:

```
map f xs =
  case xs of
    [] -> []
    (x:xs) -> f x : map f xs
```

2.3.4 Typklassen

Haskell stellt Typklassen zur Verfügung, mit denen man Operatoren überladen bzw. für verschiedene Datentypen unter gleichem Namen neu definieren kann. In einer Typklasse werden dabei die Typsignaturen der Funktionen, die ihre Mitglieder enthalten müssen, definiert, d.h. es werden die Funktionsnamen und ihr Typ angegeben. Beispiel:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Optional kann man auch die Zugehörigkeit zu anderen Typklassen voraussetzen; dies kann man als Vererbung betrachten, da die Funktionen der vererbenden Typklasse nun auch in der neuen Typklasse zur Verfügung stehen. Typklassen kann man somit in gewisser Hinsicht mit Klassen in objektorientierten Sprachen vergleichen (siehe auch [Tho97]), deshalb werden die Funktionen in Typklassen auch oft *Methoden* genannt.

Die Zugehörigkeit eines Datentyps zu einer Typklasse kann man durch *Instanziierung* bewirken. Listen sind z.B. wie folgt als Instanz von `Functor` definiert:

```
instance Functor [] where
  fmap = map
```

Bei einer Instanziierung müssen also die Implementierungen zu den Typsignaturen angegeben werden.

Typsynonyme können allerdings keine Instanzen von Typklassen sein[13]. Die einzige Möglichkeit zur Instanziierung ist also, sie in einen Datentyp einzupacken. Zu diesem Zweck ist auch eine Deklaration mittels `newtype` möglich, bei der nur ein einzelner einstelliger Konstruktor verwendet werden darf. Diese spezielle Definition ermöglicht in der Implementierung der Sprache Haskell einen schnelleren Zugriff auf das Datenelement als eine `data`-Definition.

Bei den in Haskell vordefinierten Klassen (`Ord`, `Show`, usw.) kann man Datentypen auch durch eine automatische Ableitung instanziieren, dazu steht das Schlüsselwort `deriving` bereit. Beispiel:

```
data Tree a = Leaf a | Branch Tree a Tree a
  deriving (Ord,Show)
```

2.3.5 Guards

Guards[14] stellen ein gutes Mittel dar, um Fallunterscheidungen übersichtlicher zu gestalten. Es handelt sich dabei um Prädikate über den Argumenten einer Funktion, wobei für jeden Guard ein eigener Funktionsrumpf definiert werden muss – also genau wie beim Pattern Matching für jedes Pattern.

Mit Guards kann man mathematische Funktionsdefinitionen wie

$$\max(x, y) = \begin{cases} x & \text{falls } x > y \\ y & \text{sonst} \end{cases}$$

direkt in Haskell-Notation übertragen:

```
max x y
  | x > y     = x
  | otherwise = y
```

Hierbei stellt das Prädikat `otherwise` ein Synonym für `True` dar.

2.3.6 List Comprehensions

Da Listen eine besonders wichtige Datenstruktur sind, gibt es in Haskell auch ein besonderes Mittel zur Definition komplexer Listen, nämlich *List Comprehensions*[15]. Diese sind angelehnt an der mathematischen Notation für Mengen und erlauben durch Angabe eines einzigen Ausdrucks und der Definition der darin enthaltenen Variablen durch Generatoren die Spezifikation vieler Listenelemente. Die Funktion map könnte man somit auch wie folgt definieren:

---

[13] Als Haskell-Erweiterung ist dies möglich, die Instanz gilt dann aber auch für den Basistyp und alle weiteren Synonyme dieses Typs.
[14] engl. Wächter
[15] engl. *comprehension* - Verständnis

```
map f l = [f x | x <- l]
```

Der Generator `x <- l` bewirkt, dass `x` alle Werte der Liste `l` durchläuft. Mittels Pattern Matching und der Verwendung von Guards stehen dabei auch sehr komplexe Mittel zum Filtern der Ergebnisse bereit - Beispiel:

```
[ x |  (x,y) <- [(1,2),(3,2),(3,4),(5,4)], x + y < 4 ]
```

Ergebnis dieses Ausdrucks ist `[1]`. Dabei ist `x + y < 4` ein Guard, der dafür sorgt, dass die Liste keine Elemente enthält, die dieses Prädikat nicht erfüllen.

2.3.7 Monaden

Eine besonders bedeutende Typklasse ist `Monad`. Diese ist wie folgt definiert:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a

  m >> k = m >>= \_ -> k
  fail s = error s
```

Monaden (Instanzen der Klasse `Monad`) dienen dazu, Zustände oder Aktionen zu kapseln und Aktionen hintereinander auszuführen, insbesondere sind IO-Aktionen als Monaden definiert. Die Methode `return` sorgt für die Kapselung, die Methoden `>>=` („bind“) und `>>` sorgen für die Hintereinanderausführung: Der monadische Wert wird kombiniert mit einer Funktion, die einen neuen monadischen Wert zurückliefert, wobei bei `>>` der gekapselte Zustand bzw. die gekapselte Aktion der ersten Monade keine Rolle spielt. Beispiel:

```
showfile file =
  readFile file >>= (\input -> putStrLn input)
```

Die Funktion `showfile` lädt eine Datei mittels `readFile` und zeigt den eingelesenen Text mittels `putStrLn` auf dem Bildschirm an. Da die Methoden `>>=` und `>>` zu einer sequentiellen Verarbeitung führen, steht für diese eine spezielle Syntax zur Verfügung, die am imperativen Programmierstil orientiert ist: Die `do`-Notation. Damit kann man die Funktion `showfile` wie folgt definieren:

```
showfile file = do
  input <- readFile file
  putStrLn input
```

Neben monadischem IO wollen wir Monaden auch für Ausnahme-Behandlungen verwenden. Als Beispiel betrachten wir den in der Haskell-Prelude vordefinierten Datentyp `Maybe`:

```
data Maybe a = Nothing | Just a
```

Dieser kann dazu benutzt werden, um Funktionen zu definieren, deren Berechnungen fehlschlagen können bzw. die manchmal kein Ergebnis zurückliefern, beispielsweise eine Nachschlage-Operation für ein Wörterbuch. Wollen wir mehrere solcher Berechnungen hintereinander ausführen, so erleichtert uns dies `Maybe` als Instanz der `Monad`-Klasse. Die Instanziierung ist in der Prelude wie folgt definiert:

```
instance Monad Maybe where
     Just x  >>= k = k x
     Nothing >>= k = Nothing
     return        = Just
     fail s        = Nothing
```

Aufeinanderfolgende Berechnungen auf einem `Maybe`-Wert können jetzt einfach mit `>>=` hintereinander geschaltet werden. Es ist nun keine eigene Fallunterscheidung mehr zur Überprüfung, ob die Berechnung erfolgreich war, nötig, denn dies geschieht bereits implizit innerhalb der Monaden-Methode `>>=`.

Weitere Verwendungsmöglichkeiten für Monaden sind in [Wad95] beschrieben.

3 Der Lambda-Kalkül

Der λ-Kalkül wurde in den 30er Jahren von Alonso Church eingeführt, mit dem Ziel, die Berechnungsaspekte von Funktionen formal zu erfassen. Es handelt sich dabei um ein einfaches System anonymer Funktionsausdrücke, deren Reduktions- und Konversionsregeln sich auf rein syntaktische Ersetzungen der (Teil-)Ausdrücke beschränken. Trotz seiner Einfachheit ist der λ-Kalkül sehr mächtig, denn λ-Definierbarkeit ist äquivalent zur Turing-Berechenbarkeit. Der λ-Kalkül stellt die Grundlage der funktionalen Programmierung dar: Die meisten funktionalen Sprachen könnte man auch einfach als syntaktisch angereicherten λ-Kalkül mit Datentypen betrachten.

Im Folgenden werfen wir zunächst einen Blick auf die Syntax der λ-Ausdrücke, nebst einiger weiterer wichtiger Definitionen. Im Anschluss daran betrachten wir die Konversionsregeln des λ-Kalküls nach [Bar84], den wir hier klassischen λ-Kalkül nennen. Die dort beschriebene Auswertung steht aber nicht im Einklang mit tatsächlichen Implementierungen funktionaler Programmiersprachen, denn die Definition der Werte (Ausdrücke, die nicht weiter reduziert werden) erweist sich als zu streng. Eine bessere Beschreibung für nicht-strikte Sprachen liefert der „lazy“[16] λ-Kalkül von Abramsky [Abr90], welchen wir in Abschnitt 3.3 betrachten wollen. Zum Abschluss des Kapitels erfolgt in Abschnitt 3.3.4 noch ein kurzer Vergleich mit einem Call-by-Value λ-Kalkül.

3.1 Die Syntax der Lambda-Sprache und wichtige Definitionen

Definition 3.1.1. Die Sprache Λ aller λ-Terme wird definiert als $\Lambda = L(G)$ mittels der kontextfreien Grammatik

$$G = (\{Exp, V\}, Var \cup \{\lambda, ., (,)\}, Exp, P)$$

wobei

$$P = \{Exp \to V, Exp \to (\lambda V.Exp), Exp \to (Exp\ Exp)\} \cup \{V \to v \mid v \in Var\}$$

und *Var* eine abzählbare Menge von *Variablen* bezeichne.

Anmerkung: Wir benutzen für Produktionen künftig die kürzere Backus-Naur-Form:

$$Exp \quad ::= V \mid (\lambda V.Exp) \mid (Exp\ Exp)$$

$$V \quad :: = v \in Var$$

[16] Sharing wird hier nicht betrachtet, obwohl der Name dies suggeriert

Außerdem verzichten wir auf die Angabe der Terminale und Nichtterminale, da diese eindeutig aus den Produktionen hervorgehen. Das Startsymbol soll das in der ersten Produktion definierte Nichtterminal sein.

Namenskonvention: Mit den Buchstaben *r, s, t* bezeichnen wir im Folgenden λ-Terme und mit *x, y, z* Variablen aus *Var*.

Terme der Form $(\lambda x.\ t)$ bezeichnen wir als *Abstraktion* oder *Funktionsausdruck*, Terme der Form $(s\ t)$ als *Applikation* oder *Anwendung*. Im Umgang mit Klammern wollen wir eine implizite Linksklammerung annehmen, so dass $r\ s\ t$ dasselbe ist wie $((r\ s)\ t)$, ebenso lassen wir ggf. die Klammern bei Abstraktionen weg, wobei Abstraktionen so weit nach rechts wie möglich aufgefasst werden, so dass $\lambda x.s\ t$ dasselbe bedeutet wie $(\lambda x.(s\ t))$. Für Ausdrücke der Form $(\lambda x_1.(\lambda x_2.\ \ldots\ (\lambda x_n\ .\ t)))$ schreiben wir auch abkürzend $\lambda x_1\ x_2\ \ldots\ x_n\ .\ t$ oder gleich $\lambda x_1 \ldots x_n\ .\ t$, falls wir (wie hier) Indizes benutzen.

Unterterme definieren wir in naheliegender Weise über die Struktur der Terme:

Definition 3.1.2. Die Menge *Subt*(*t*) der *Unter-* bzw. *Teilterme* eines Terms *t*, wird induktiv[17] definiert durch:

$Subt(x) = \{x\}$, falls $x \in Var$

$Subt(\lambda x.t) = Subt(t) \cup \{\lambda x.t\}$

$Subt(s\ t) = Subt(s) \cup Subt(t) \cup \{(s\ t)\}$

Abstraktionen können Variablen *binden*: Im Ausdruck $\lambda x.\ t$ ist die Variable *x* gebunden im Gültigkeitsbereich (Skopus) *t.* Variablen, die nicht gebunden sind, sind *frei*. Wir formalisieren diesen Zusammenhang mit den folgenden Definitionen:

Definition 3.1.3. Die Menge *FV*(*t*) der freien Variablen des Terms *t* wird induktiv definiert durch:

$FV(x) = \{x\}$, falls $x \in Var$

$FV(\lambda x.\ t) = FV(t) \setminus \{x\}$

$FV(s\ t) = FV(s) \cup FV(t)$

Definition 3.1.4. Die Menge *BV*(*t*) der gebundenen Variablen des Terms *t* wird induktiv definiert durch:

$BV(x) = \emptyset$, falls $x \in Var$

$BV(\lambda x.\ t) = \{x\} \cup BV(t)$

[17] d.h. als die kleinste Menge, die die Bedingungen erfüllt

$$BV(s\ t) \qquad = BV(s) \cup BV(t)$$

Diese Definition sagt jedoch nichts über die Beziehung der gebundenen Variablen zu ihren Bindungsbereichen aus. Durch Verwendung desselben Variablennamens kann es zu Konflikten kommen; in diesen Fällen gilt: Der innerste Bindungsbereich zählt. Um Konflikte durch gleiche Variablennamen zu vermeiden, kann man die gebundenen Variablen umbenennen und so die verschiedenen Bindungsbereiche sichtbar machen, da es nur auf die Verbindung zwischen dem gebundenen Namen und dem Vorkommen ankommt.

Beispiel 3.1.5. $\lambda x.(x\ \lambda x.\ x)$ wird durch Umbenennung zu $\lambda x.(x\ \lambda z\ .\ z)$

Weitere Betrachtungen zu Umbenennungen folgen in Abschnitt 3.2.

Definition 3.1.6. Als Menge der *geschlossenen* λ-Terme bezeichnen wir

$$\Lambda_0 = \{t \in \Lambda \mid FV(t) = \varnothing\}.$$

Einen Term, der nicht geschlossen ist (also freie Variablen enthält) nennen wir *offen*, einen geschlossenen Term nennen wir auch *Kombinator* oder *Programm.*

Definition 3.1.7. Wir definieren die folgenden Synonyme für Kombinatoren:

$$I \equiv \lambda x.\ x$$

$$K \equiv \lambda x\ y.\ x$$

$$S \equiv \lambda x\ y\ z\ .\ x\ z\ (y\ z)$$

$$Y \equiv \lambda x.(\lambda y.\ x\ (y\ y))\ (\lambda y.\ x\ (y\ y))$$

$$\Omega \equiv (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

Mit dem Symbol ≡ möchten wir dabei syntaktische Gleichheit ausdrücken.

3.2 Der klassische Lambda-Kalkül

Wir wollen nun die Kalkülregeln des klassischen λ-Kalküls nach [Bar84] betrachten, die eine Gleichheitsrelation für λ-Terme definieren. Dazu brauchen wir zunächst den Begriff der Substitution:

Definition 3.2.1. Die Substitution, die die freien Vorkommen einer Variablen x in einem Term s durch einen Term t ersetzt, geschrieben $s\{t / x\}$[18], wird induktiv definiert durch:

[18] Übliche Schreibweisen sind auch s{x:=t}, s[t / x] oder s[x:=t].

$$x\{t / x\} = t$$

$$y\{t / x\} = y$$

$$(r\ s)\{t / x\} = (r\{t / x\}\ s\{t / x\})$$

$$(\lambda x.\, s)\{t / x\} = (\lambda x.\, s)$$

$$(\lambda y.\, s)\{t / x\} = \begin{cases} (l\, y.s\{t / x\}), \text{ falls } x \notin FV(s) \vee y \notin FV(t) \\ (l\, z.s\{z / y\})\{t / x\}, \text{ sonst} \quad (z \text{ neuer Variablenname}) \end{cases}$$

Bei Substitutionen muss man darauf achten, dass dadurch nicht ungewollt Variablen von Lambda-Ausdrücken eingefangen werden. Dies geschieht, wie in der Definition gesehen, durch Umbenennung.

Beispiel 3.2.2. $(\lambda x.\, x\, y)\{x / y\} = (\lambda z.\, z\, y)\{x / y\} = \lambda z.\, z\, x.$

Im Folgenden wollen wir ähnliche Situationen vereinfachen, indem wir annehmen, dass die gebundenen Variablen stets so gewählt (bzw. im Vorhinein umbenannt) werden, dass sie von den freien Variablen verschieden sind.

Mittels Substitution können wir die beiden elementaren Umformungen definieren: a-Konversion und b-Konversion. Die a-Konversion ist dabei nichts weiter als die Umbenennung der gebundenen Variablen und wird in der Literatur oftmals implizit durchgeführt, ohne dies gesondert zu erwähnen, denn es kommt, wie bereits erwähnt, nur auf die Verbindung zwischen dem gebundenen Namen und dem Vorkommen an. Will man die Umbenennung formal betrachten, so kann man dies mittels der folgenden Regel tun:

Definition 3.2.3. Der *umittelbare Umbenennungsschritt* $\underset{a}{\rightarrow}$ ist definiert durch:

$$(a) \qquad (\lambda x.\, t) \underset{a}{\rightarrow} (\lambda y.\, t\{y / x\}), \text{ falls } y \notin FV(t)$$

Eine Umbenennung ist nun die Ausführung unmittelbarer Umbenennungsschritte in beliebigen Untertermen – die genaue Definition folgt im nächsten Unterabschnitt. Wir werden Umbenennungen für die Definition der Gleichheit des klassischen λ-Kalküls formal berücksichtigen, danach werden wir jedoch Umbenennungen implizit durchführen und Ausdrücke, die bis auf Umbenennung der gebundenen Variablen gleich sind, sollen dann auch als syntaktisch gleich bezüglich ≡ gelten (a-Gleichheit).

Die interessantere Umformung ist die b-Konversion, die bei impliziter Umbenennung ausreicht, um die Gleichheit des klassischen λ-Kalküls zu beschreiben. Der grundlegende Schritt der b-Konversion ist die b-Kontraktion:

Definition 3.2.4. Die *b-Kontraktion* $\underset{b}{\rightarrow}$ ist definiert durch:

$$(b) \qquad (\lambda x.\, s)\ t \underset{b}{\rightarrow} s\{t / x\}$$

Ein Term der Form $r \equiv (\lambda x.s)\ t$ heißt β-*Redex*[19] und r' mit $r \underset{\beta}{\rightarrow} r'$ das dazugehörige *Kontraktum.*

3.2.1 Ersetzbarkeit und Kongruenz-Begriff

Wir wollen mit den beiden Umformungsschritten (α) und (β) eine Gleichheit entwickeln, die Ersetzbarkeit garantiert. Dazu erlauben wir die Durchführung dieser Umformungsschritte in beliebigen Untertermen. Dies lässt sich am einfachsten mit Hilfe von Kontexten formalisieren. Kontexte sind dabei Termkonstrukte mit einem Loch [] als Unterterm. Analog zu Untertermen definieren wir diese über die Struktur der Terme:

Definition 3.2.5. *Termkontexte,* kurz *Kontexte,* sind induktiv definiert durch:

- [] ist ein Kontext.
- Sei $C[] \in \Lambda$ ein beliebiger Kontext: Dann sind auch $\lambda x.C[]$, $C[]\ t$ und $t\ C[]$ wieder Kontexte, für beliebige $t \in \Lambda$, $x \in Var$.

Für einen Kontext $C[]$ soll $C[] \in \Lambda$ bedeuten, dass $C[]$ ein Kontext über Λ ist, d.h $\forall\ t \in \Lambda \Rightarrow C[t] \in \Lambda$, wobei $C[t]$ die Einsetzung des Terms t anstelle des Loches beschreibt. [] ist der leere Kontext, setzt man in diesen einen Ausdruck ein, so ergibt sich der Ausdruck selbst.

Anmerkung: Wir benutzen für Kontext-Definitionen künftig die kürzere Produktionen-Schreibweise:

$$C ::= [] \mid \lambda x.C \mid C\ t \mid t\ C$$

Mit Kontexten haben wir ein sehr praktisches Mittel zur Hand, um Aussagen über beliebige Unterterme zu treffen. Insbesondere können wir auch die Ausführung unserer Umformungsschritte in Untertermen mittels Kontexten beschreiben:

Definition 3.2.6. Wir definieren den *Umbenennungsschritt* $\underset{\alpha, C}{\rightarrow}$ wie folgt:

$$\forall\ C[] \in \Lambda: C[s] \underset{\alpha, C}{\rightarrow} C[t] \text{ gdw. } s \underset{\alpha}{\rightarrow} t$$

Eine beliebige Folge von Umbenennungsschritten bzw. die reflexiv-transitive Hülle $\underset{\alpha, C}{\rightarrow}^{*}$ bezeichnen wir als *Umbenennung* bzw. α-*Konversion* und benutzen das Symbol $=_\alpha$.

Somit haben wir unseren intuitiv eingeführten Begriff der α-Gleichheit formal präzisiert. Durch unsere Vereinbarung, die Umbenennung implizit zu betrachten, entspricht die syntaktische Gleichheit $\equiv$ also der α-Konversion $=_\alpha$.

[19] redex: Abk. für reducible expression, also „reduzierbarer Ausdruck"

Da sich ein Umbenennungsschritt durch erneute Umbenennung mit dem ursprünglichen Namen rückgängig machen lässt, gilt:

Aussage 3.2.7. $=_a$ ist auch symmetrisch und somit eine Äquivalenzrelation.

Definition 3.2.8. Wir definieren den *b-Reduktionsschritt* $\underset{b,C}{\rightarrow}$ wie folgt:

$$\forall\, C[] \in \Lambda\colon C[s] \underset{b,C}{\rightarrow} C[t] \text{ gdw. } s \underset{b}{\rightarrow} t$$

Eine beliebige Folge von *b-Reduktionsschritten* bzw. die reflexiv-transitive Hülle $\underset{b,C}{\rightarrow}^*$ bezeichnen wir als *b-Reduktion* und das Ergebnis einer *b-Reduktion* als *Redukt*.

Die Umkehrung eines *b*-Reduktionsschrittes bezeichnen wir als *b-Expansionsschritt* und die Umkehrung der *b*-Reduktion als *b-Expansion*[20]. Die Ausführung entweder eines *b*-Reduktionsschrittes oder aber eines *b-Expansionsschrittes*, d.h. die symmetrische Hülle $\underset{b,C}{\leftrightarrow}$, nennen wir *b-Konversionsschritt*. Eine beliebige Folge von *b-Konversionsschritten*, d.h. die reflexiv-transitive Hülle $\underset{b,C}{\leftrightarrow}^*$ bezeichnen wir als *b-Konversion* und benutzen das Symbol $=_b$.

Zu beachten ist, dass die *b*-Konversion nicht der symmetrischen Hülle der *b*-Reduktion entspricht, denn diese beschreibt nur entweder eine *b*-Reduktion oder eine *b*-Expansion, erlaubt also nicht die Kombination verschiedener *b*-Reduktionen oder *b*-Expansionen, wie dies bei der *b*-Konversion der Fall ist.

Wir werden uns in Abschnitt 3.2.2 genauer mit der *b*-Reduktion und der dadurch beschriebenen Auswertung beschäftigen. Zunächst aber wollen wir unsere Betrachtungen zum Gleichheitsbegriff im klassischen Lambda-Kalkül zu Ende führen. Dazu betrachten wir die formale Kombination der *a*- und *b*-Konversion:

Definition 3.2.9. Sei $s \underset{a+b,C}{\leftrightarrow} t$ entweder ein Umbenennungsschritt oder ein *b*-Konversionsschritt – wir nennen dies einen *a,b-Konversionsschritt*: Dann ist die *a,b-Konversion* gegeben durch eine Folge von *a,b*-Konversionschritten bzw. der reflexiv-transitiven Hülle $\underset{a+b,C}{\leftrightarrow}^*$ – wir benutzen das Symbol $=_\lambda$.

Mit $=_\lambda$ steht uns also der auf den Regeln der *a*- und *b*-Konversion aufgebaute Gleichheitsbegriff zur Verfügung, dieser entspricht der *b*-Konversion mit impliziter *a*-Gleichheit. Falls $s =_\lambda t$, so nennen wir *s* und *t konvertibel*. Wir schreiben abkürzend auch einfach = statt $=_\lambda$, wenn der Zusammenhang klar ist.

[20] Diese Bezeichnung wird in der Literatur zum Teil auch für die Umkehrung der *b*-Kontraktion verwendet.

Zur formalen Betrachtung der Ersetzbarkeit steht der Kongruenzbegriff zur Verfügung, dieser ist ebenfalls mittels Kontexten definiert:

Definition 3.2.10. Eine Äquivalenzrelation $\sim \subseteq L \times L$ ist eine *Kongruenz* gdw.

$$\forall\, s, t, C[] \in L\colon s \sim t \Rightarrow C[s] \sim C[t].$$

Erfüllt eine partielle Ordnung diese Eigenschaft, so nennen wir sie *Präkongruenz*.

Ist unsere Gleichheitsrelation also eine Kongruenz, so bleibt die Gleichheit zweier Ausdrücke erhalten, wenn man sie in einen beliebigen Kontext einsetzt und die daraus entstehenden Ausdrücke betrachtet. Somit können gleiche Unterterme gegeneinander ausgetauscht werden. Es ist leicht zu sehen, dass aufgrund unserer Definition die Bedingung $s = t \Rightarrow C[s] = C[t]$ erfüllt ist, da diese Eigenschaft auch für die symmetrische bzw. transitiv-reflexive Hülle erhalten bleibt. (Man könnte = auch einfach als Kongruenzabschluss[21] der Kombination der beiden Regeln (a) und (b) definieren.) Die b-Reduktion ist eine Präkongruenz (deswegen wird in der Literatur für den Begriff der Präkongruenz teilweise auch „Reduktionsrelation“ verwendet).

3.2.2 Auswertung

Aus operativer Sicht interessieren wir uns für die Auswertung mittels b-Reduktion zu einem Wert. Da Ausdrücke oftmals mehrere b-Redexe enthalten, gibt es natürlich auch unterschiedliche Reduktionsfolgen (vgl. Abschnitt 2.1.3). Wir wollen uns zunächst davon überzeugen, dass es, bei zunächst unterschiedlich verlaufenen Reduktionen eines Ausdrucks, jederzeit die Möglichkeit gibt, unterschiedliche Zwischenergebnisse zu einem gemeinsamen (d.h. syntaktisch gleichen) Ergebnis zu reduzieren. Diese Eigenschaft nennt sich *Konfluenz* und garantiert, dass bei einer Auswertung kein Backtracking nötig ist, d.h. man muss niemals Reduktionsschritte mittels b-Expansion rückgängig machen, um zu einem möglichem Ergebnis zu gelangen. Die formale Definition lautet wie folgt:

Definition 3.2.11. Eine Relation $\rightarrow \subseteq L \times L$ ist *konfluent* gdw.

$$\forall\, r, s_1, s_2 \in L\colon r \rightarrow^* s_1 \wedge r \rightarrow^* s_2 \Rightarrow \exists\, t \in L\colon s_1 \rightarrow^* t \wedge s_2 \rightarrow^* t.$$

Satz 3.2.12. Die b-Reduktion ist konfluent.

Beweis: Siehe z.B. [Tak95].

Eine andere wünschenswerte Eigenschaft ist, dass konvertible Ausdrücke durch Reduktion in einen gemeinsamen Ausdruck überführt werden können:

Definition 3.2.13. Eine Relation $\rightarrow \subseteq L \times L$ hat die *Church-Rosser-Eigenschaft* gdw.

$$\forall\, r, s, t \in L\colon r \leftrightarrow^* s \Leftrightarrow \exists\, t\colon r \rightarrow^* t \,{}^*\!\leftarrow s.$$

[21] Gemeint ist die Erweiterung $s \rightarrow t \Rightarrow C[s] \rightarrow C[t]$ und Bildung der äquivalenten Hülle in einem Schritt.

Dies ist der Fall, da die Konfluenz und die Church-Rosser-Eigenschaft äquivalent sind:

Satz 3.2.14. Eine Relation $\rightarrow$ ist konfluent gdw. sie die Church-Rosser-Eigenschaft hat.

Beweis: $\Rightarrow$: Die Behauptung folgt durch wiederholte Ausnutzung der Konfluenz-Eigenschaft. $\Leftarrow$: Klar.

Nun stellt sich natürlich die Frage, wie unsere Werte aussehen sollen, also: Welche Ausdrücke werden nicht weiter reduziert?

Der erste Ansatz ist, die Ausdrücke als Werte aufzufassen, die nicht weiter reduziert werden können, weil sie keinen β-Redex enthalten:

Definition 3.2.15. Enthält ein Term $t \in \Lambda$ keinen β-Redex, so ist t eine *Normalform*, kurz NF. Ein Term $t \in \Lambda$ *hat* eine Normalform, gdw. $\exists\, s$: $t = s$ und s ist eine NF.

Beispiel 3.2.16. Terme und deren Beziehung zur Normalform:

- $I \equiv \lambda x\,.\,x$ ist eine NF.
- $K\,I \equiv (\lambda x\,y\,.\,x)\ \lambda x\,.\,x$ hat eine NF, da $K\,I = \lambda y\,.\,I$
- $\Omega \equiv (\lambda x\,.\,x\ x)\ (\lambda x\,.\,x\ x)$ hat keine NF, da durch β-Kontraktion wieder der gleiche Ausdruck entsteht. Die Reduktion von Ω zu einer NF terminiert somit nicht, aufgrund der Konfluenz ist auch kein anderer Weg zu einer NF (der zunächst β-Expansion benutzt) möglich.

Normalformen haben die schöne Eigenschaft, dass sie eindeutig sind, d.h. jede terminierende Reduktionsfolge konvertibler Ausdrücke endet mit der gleichen NF (modulo α):

Satz 3.2.17. Falls $s = t$, $s \underset{\beta,C}{\rightarrow}{}^{*}\ r \wedge t \underset{\beta,C}{\rightarrow}{}^{*}\ r'$ und r und r' sind NFs, dann ist $r \equiv r'$.

Beweis: Folgt aus der Konfluenz bzw. Church-Rosser-Eigenschaft.

Sieht man Normalformen jedoch als Bedeutung von λ-Termen an und setzt somit alle Terme, die keine NF haben, gleich, so führt dies zu einer inkonsistenten Theorie, d.h. alle Formeln $s = t$ mit geschlossenen Termen s und t können bewiesen werden.

Beispiel 3.2.18. $\lambda x.(x\ K\ \Omega)$ und $\lambda x.(x\ S\ \Omega)$ haben keine NF, da Ω keine NF hat. Somit wäre bei Identifizierung der Terme ohne NF nun

$$\lambda x.(x\ K\ \Omega) = \lambda x.(x\ S\ \Omega)$$

und damit auch

$$\lambda x.(x\ K\ \Omega)\ K = K\ K\ \Omega = K = S = K\ S\ \Omega = \lambda x.(x\ S\ \Omega)\ K.$$

Analog zu diesem Beispiel könnte man die Gleichheit zweier beliebiger anderer Programme zeigen. Da jetzt alle Kombinatoren gleich wären, ginge jegliche Aussagekraft verloren. Wir benötigen daher eine andere Definition für Werte:

Definition 3.2.19. Ein Term $t \in \Lambda$ ist eine *Kopfnormalform*[22], kurz HNF, gdw.

$$t \equiv (\lambda x_1 ... x_n \,.\, (y\ t_1 ... t_k)),$$

wobei $x_i, y \in Var$, $t_j \in \Lambda$, für $n \geq 0$, $k \geq 0$, $0 \leq i \leq n$, $0 \leq j \leq k$.

Ein Term $t \in \Lambda$ *hat* eine Kopfnormalform, gdw. $\exists\, s$: $t = s$ und s ist eine HNF.

In unserem Beispiel ist $\lambda x.(x\ K\ \Omega)$ also eine HNF und somit gilt $\lambda x.(x\ K\ \Omega) \neq \lambda x.(x\ S\ \Omega)$, da $K \neq S$. Es ist konsistent, Terme ohne HNF als unlösbar bzw. undefiniert zu betrachten und somit gleichzusetzen, denn es gilt folgender Satz:

Satz 3.2.20. Sei s ein Term ohne HNF und t eine NF, dann gilt für einen beliebigen Kontext $C[\,]$:

$$C[s] = t \Rightarrow (\forall r \in \Lambda)\ C[r] = t$$

Beweis: Siehe [Bar84, Seite 374].

Terme ohne HNF bezeichnet man in diesem Zusammenhang auch als *bedeutungslose* Terme. Allerdings sind HNFs nicht eindeutig, d.h. ein Term kann mehrere HNFs haben, die nicht syntaktisch gleich sind. Selbstverständlich können diese unterschiedlichen HNFs zu einer gemeinsamen HNF reduziert werden (d.h. diese HNFs sind gleich im Sinne von konvertibel), aber es gibt keine einfache Auswertungsstrategie, die dies leistet. Aus diesem Grund ist es üblich, das Ergebnis der sogenannten Kopfreduktion als Repräsentanten der HNFs eines Terms zu wählen. Die Kopfreduktion entspricht der Normalordnungsreduktion, außer dass die Kopfredexe bevorzugt ausgewertet werden. Ein Kopfredex ist dabei so definiert, dass falls r ein Redex ist, dann ist r ein Kopfredex in einem Term der Form $(\lambda x_1 ... x_n.(r\ t_1 ... t_k)))$. Wie die Normalordnungsreduktion terminiert auch die Kopfreduktion immer, wenn ein Term eine HNF hat (für Details siehe [Bar84]).

3.2.3 Extensionalität

Im Umkehrschluss zu $s = t \Rightarrow s\ x = t\ x$, was ja aus der Ersetzbarkeit folgt, würden wir für Funktionen auch gerne folgern können, dass diese gleich sind, falls sich diese auf allen möglichen Argumenten gleich verhalten, also: $s\ x = t\ x \Rightarrow s = t$ für $x \notin FV(s\ t)$. Dies kann jedoch mit unserer bisherigen Gleichheitsrelation nicht bewiesen werden, ebenso wenig können wir zeigen, dass $(\lambda x.s\ x) = s$ ist, obwohl wir bei jeder Anwendung auf ein beliebiges Argument t die entstehenden Ausdrücke durch b-Kontraktion ineinander überführen können: $(\lambda x.s\ x)\ t \rightarrow_b s\ t$. Wollen wir also Funktionsausdrücke, die sich auf

[22] engl. *Head Normal Form*

gleichen Argumenten gleich verhalten, als gleich betrachten, so müssen wir eine der beiden folgenden Regeln in unser Kalkül aufnehmen:

(*h*) $(\lambda x.\, t\ x) \underset{h}{\rightarrow} t$, wobei $x \notin FV(t)$

(ext) $s\ x = t\ x \Rightarrow s \underset{\text{ext}}{\rightarrow} t$, wobei $x \notin FV(s\ t)$

Diese erweiterten Theorien werden mit λ + ext oder λh bezeichnet, wobei diese beiden äquivalent sind (Beweis: siehe [Bar84, Seite 32]), so dass man durch Hinzunahme der einen Regel auch die Gleichheiten herleiten kann, die direkt aus der anderen Regel folgen würden. Wir können nun aus der Gleichheit als Funktion – d.h. Anwendung auf gleiche Argumente liefert die gleichen Ergebnisse – die Gleichheit der Ausdrücke folgern. Diese Eigenschaft heißt *Extensionalität*.

3.3 Der „lazy“ Lambda-Kalkül

Der klassische λ-Kalkül stellt zwar eine konsistente Theorie dar, die auch einige interessante Aussagen ermöglicht, z.B. über die Eigenschaften verschiedener Auswertungsstrategien. Er ist jedoch nicht so gut geeignet, um die Auswertung in existierenden funktionalen Programmiersprachen zu beschreiben, denn diese werten Funktionsausdrücke nicht aus. Kopfnormalformen stellen somit nicht die passende Definition für Werte dar, vielmehr erweisen sich *schwache* Kopfnormalformen als geeignete Werte:

Definition 3.3.1. Ein Term $t \in \Lambda$ ist eine *schwache Kopfnormalform*[23], kurz WHNF, gdw. $t \equiv \lambda x.\, t'$ für ein $t' \in \Lambda$.

Ein Term $t \in \Lambda$ *hat* eine WHNF, gdw. $\exists\ s$: $t = s$ und s ist eine WHNF.

Im Zusammenhang mit dem Lambda-Kalkül meinen wir von nun an, wenn wir von einem Wert sprechen, immer eine WHNF. Gegenüber der Verwendung von HNF-Werten ergeben sich dadurch bedeutende Änderungen: So enthält zum Beispiel $\lambda x.((\lambda y.y)\ t)$ den Kopf-Redex $((\lambda y.y)\ t)$, dieser Ausdruck ist also keine HNF, wohl aber eine WHNF. Falls dieser Kopf-Redex ein nicht-terminierender Ausdruck ist, bedeutet dies, dass die Reduktion zur HNF nicht terminiert, während bei einer WHNF-Auswertung der Kopf-Redex nicht reduziert wird und die Berechnung sofort terminiert, da bereits ein Wert vorliegt.

Abramskys „lazy“ λ-Kalkül beschreibt die Auswertung nun mittels einer operationalen Semantik, die auf WHNFs basiert. Zudem schlägt Abramsky einen anderen Gleichheitsbegriff vor, der nicht über eine Herleitung anhand der Gleichheitsregeln, sondern vielmehr über das *Verhalten* zweier Ausdrücke bei Einsetzung in einen Programmkontext definiert ist:

[23] engl. *Weak Head Normal Form*

Zwei geschlossene Terme $s, t \in \Lambda_0$ sind gleich, falls es keinen Kontext $C[]$ gibt, so dass sich $C[s]$ anders verhält als $C[t]$. Das Verhalten, welches wir in einem Gedankenexperiment beobachten wollen, ist die Terminierung, d.h. die erfolgreiche Auswertung zu einem Wert. Dieses Verfahren ist natürlich nicht effektiv, da es unendlich viele Kontexte gibt und das Halteproblem unentscheidbar ist, aber die darauf basierende Definition der Verhaltensgleichheit ist wesentlich flexibler als die Herleitung über Gleichheitsregeln. Insbesondere bleibt der Gleichheitsbegriff auch bei veränderter Auswertung bestehen (dies ändert nur das beobachtete Verhalten), während beim klassischen λ-Kalkül die Hinzunahme neuer Regeln bzw. Axiome nötig wäre und so die Gleichheit neu definiert werden müsste. Allerdings kann sich die Gleichheit jetzt durch einen erweiterten bzw. verringerten Sprachumfang (bei gleichen Auswertungsregeln) ändern, da dies die Betrachtung über alle Kontexte beeinflusst; im Gegensatz dazu bleibt die Gleichheit im klassischen λ-Kalkül in diesem Fall erhalten. Es stellt sich außerdem heraus, dass die Verhaltensgleichheit des „lazy“ λ-Kalküls mehr gleiche Ausdrücke identifiziert, als die Gleichheit des klassischen λ-Kalküls, dies beeinträchtigt aber die Aussagekraft nicht, da eben nur die Ausdrücke gleich sind, die durch die Beobachtung der Reduktion nicht unterschieden werden können. Mehr zu der Verhaltensgleichheit des „lazy“ Lambda-Kalküls folgt in Abschnitt 3.3.1.

Zunächst wollen wir uns der Auswertung widmen. Wir definieren dazu die operationale Semantik zunächst als „big-step“ Semantik mittels einer Auswertungsrelation, die angibt, zu welchem Wert ein Programm *konvergiert*, sofern die Auswertung möglich ist:

Definition 3.3.2. Die Relation $t \Downarrow s$ („t konvergiert zum Wert s“) wird induktiv über Λ_0 wie folgt definiert:

$$\lambda x.t \Downarrow \lambda x.t$$

$$\frac{t_1 \Downarrow \lambda x.s_1 \quad r\{t_2 / x\} \Downarrow s_2}{t_1\, t_2 \Downarrow s_2}$$

(Die Schreibweise soll ausdrücken: falls $t_1 \Downarrow \lambda x.s$ und $r\{t_2 / x\} \Downarrow s_2$, dann $t_1\, t_s \Downarrow s_2$.)

$t \Downarrow$ („t konvergiert“), falls $\exists\, s$: $t \Downarrow s$. $t \Uparrow$ („t divergiert“), falls $\neg(t \Downarrow)$.

Eine entsprechende „small step“ Semantik wird anhand einer Ein-Schritt-Reduktion $\rightarrow$ definiert:

$$\frac{s \rightarrow s'}{s\, t \rightarrow s'\, t}$$

$$(l\, x.s)\, t \rightarrow s\{t / x\}$$

Die zugehörige Auswertungsrelation wird dann definiert durch: $s \downarrow t$, falls $s \rightarrow^* t$ und t ist eine WHNF. Es gilt: $s \downarrow t$ gdw. $s \Downarrow t$. Wir zeigen dieses Ergebnis in Kapitel 5 für verallgemeinerte Varianten von $\downarrow$ und $\Downarrow$.

Die Definition von $\rightarrow$ entspricht der Beschränkung der b-Reduktionsschritte auf Reduktionskontexte. Reduktionskontexte sind dabei:

$$R ::= [] \mid R\ t$$

Wie man sieht, handelt es sich hierbei um die Normalordnungs-Auswertungsstrategie: Ein Ausdruck kann entweder unmittelbar reduziert werden, oder der Normalordnungsredex befindet sich auf der am weitesten links stehenden linken Seite einer Anwendung.

3.3.1 Verhaltensgleichheit

Wir wollen nun die bereits angedeutete Verhaltensgleichheit formal definieren. Als Grundlage benutzen wir dazu eine kontextuelle Ordnungsrelation, welche im Prinzip beschreibt, welche (Unter-)Ausdrücke seltener zu terminierenden Programmen führen:

Definition 3.3.3. Die *operationale* bzw. *kontextuelle Approximation* $\leq_c \subseteq \Lambda_0 \times \Lambda_0$ wird definiert durch:

$$s \leq_c t \Leftrightarrow \forall\ C[] \in \Lambda_0 \colon C[s] \Downarrow \Rightarrow C[t] \Downarrow$$

Der Ausdruck s ist im Sinne der kontextuellen Approximation kleiner als t, falls immer dann, wenn ein Ausdruck mit Unterausdruck s konvergiert, auch der entsprechende Ausdruck mit t statt s als Unterausdruck konvergiert.

Man kann $\leq_c$ wie folgt auf Λ erweitern:

$$s \leq_c t \Leftrightarrow \forall (\sigma : V \rightarrow \Lambda_0) \colon \sigma(s) \leq_c \sigma(t)$$

wobei $\sigma(s)$ die Anwendung der Substitution $\sigma(x)$ auf jedes $x \in FV(s)$ in s bezeichnet.

Man erhält also die Erweiterung auf offene Terme, indem man freie Variablen gleichmäßig durch geschlossene Terme ersetzt und dabei alle Möglichkeiten dieser Ersetzung betrachtet.

Somit kann die kontextuelle Gleichheit für beliebige Terme angeben werden:

Definition 3.3.4. Die *operationale Äquivalenz* bzw. *kontextuelle Gleichheit* $=_c$ wird definiert durch:

$$s =_c t \Leftrightarrow s \leq_c t \wedge t \leq_c s$$

Wie leicht zu sehen ist, ist $=_c$ eine Kongruenz, Ersetzbarkeit ist also gewährleistet.

3.3.2 Bisimulation

Die Definition der kontextuellen Gleichheit ist allerdings recht unpraktisch, da es schwierig ist, Aussagen über alle Kontexte zu machen. Ein schrittweises Vorgehen erlaubt hingegen die (applikative) Bisimulation, die mittels einer Kette $\leq_{b,k}$ von Relationen über Λ_0 definiert ist:

Definition 3.3.5. Wir definieren $\leq_{b,k} \subseteq \Lambda_0 \times \Lambda_0$ induktiv wie folgt:

$$\leq_{b,0} = \Lambda_0 \times \Lambda_0$$

$$s \leq_{b,k+1} t \Leftrightarrow (s \Downarrow \lambda x.s_1 \Rightarrow \exists\, t_1\colon t \Downarrow \lambda y.t_1 \wedge \forall r \in \Lambda_0\colon s_1\{r / x\} \leq_{b,k} t_1\{r / y\})$$

Die Relation $\leq_b$ ist wie folgt definiert:

$$s \leq_b t \Leftrightarrow \forall k \geq 0\colon s \leq_{b,k} t$$

Wir bezeichnen $\leq_b$ als Simulation und $=_b$ mit $s =_b t \Leftrightarrow s \leq_b t \wedge t \leq_b s$ als Bisimulation.

Man kann $\leq_b$ als Beobachtung eines mehrstufigen Experiments beschreiben:

Stufe 0: Bevor die erste Beobachtung gemacht wird, hat man noch keine Möglichkeit, zwei geschlossene Terme $s, t \in \Lambda_0$ zu differenzieren, daher nehmen wir zunächst an, dass $s \leq_b t$ gilt.

Stufe 1: Wir werten nun s aus. Falls s divergiert, so gilt $s \leq_b t$ ohne Einschränkung. Falls s konvergiert, so muss auch t konvergieren, damit $s \leq_b t$ gelten kann. Wir wissen jedoch sonst nichts über die entstandenen Werte und setzen das Experiment somit auf Stufe 2 fort.

Stufe 2: Wir wissen, dass $s \Downarrow \lambda x.s_1$. Wir versorgen den Ausdruck mit einem neuen Argument, d.h. wir betrachten $(\lambda x.s_1)\ r$ bzw. $s_1\{r / x\}$. Divergiert dieser Ausdruck, so gilt $s \leq_b t$ ohne Einschränkung. Konvergiert dieser Ausdruck, so versorgen wir auch t bzw. dessen Wert mit einem neuen Argument und überprüfen, ob auch dieser Ausdruck konvergiert und setzen unser Experiment gegebenenfalls mit der nächsten Stufe fort.

In weiteren Stufen versorgen wir die Ausdrücke mit weiteren Argumenten und überprüfen analog zu den vorherigen Schritten die Konvergenz der entstehenden Ausdrücke.

Der Experimentator kann also in jeder Stufe nur das Verhalten der Konvergenz beobachten, aber nicht sagen, welche Teilterme die Abstraktion enthält. Wie die kontextuelle Approximation $\leq_c$ beschreibt auch die Relation $\leq_b$, welche (Unter-)Ausdrücke seltener zu terminierenden Programmen führen. In der Tat sind die Relationen $\leq_c$ und $\leq_b$ äquivalent:

Satz 3.3.6. Es gilt: $\leq_c = \leq_b$.

Beweis: Spezialfall von [Abr90, Proposition 6.6]

3.3.3 Weitere Eigenschaften

Wir notieren unseren Gleichheitsbegriff im „lazy“ λ-Kalkül von nun an mit dem Symbol $=_{\lambda l}$, ebenso verwenden wir für unsere Ordnungsrelation das Symbol $\leq_{\lambda l}$. Die formale Theorie des „lazy“ λ-Kalküls können wir somit darstellen als $\lambda l = (\Lambda, \leq_{\lambda l}, =_{\lambda l})$.

Wir wollen noch kurz einige weitere interessante Eigenschaften des „lazy“ λ-Kalküls bzw. Resultate aus [Abr90] betrachten:

Die folgenden Ergebnisse stammen aus [Abr90, Proposition 2.7]:

Resultat: Der klassische λ-Kalkül ist im „lazy“ λ-Kalkül enthalten, d.h.

$$s =_\lambda t \Rightarrow s =_{\lambda l} t.$$

Resultat: Ω ist ein kleinstes Element bzgl. $\leq_{\lambda l}$.

Resultat: $Y\,K$ ist ein größtes Element bzgl. $\leq_{\lambda l}$.

Resultat: Es gilt folgende eingeschränkte Extensionalität:

$$\lambda x.\, t\, x =_{\lambda l} t, \forall\, t \Downarrow, x \notin FV(t).$$

Wie man sieht, kann man die Extensionalität in λl für terminierende Terme folgern, während dies im klassischen λ-Kalkül nicht so ohne weiteres möglich ist. λ+ext ist jedoch nicht in λl enthalten, da z.B. $\lambda x.\Omega\, x =_{\lambda+\text{ext}} \Omega$, aber $\lambda x.\Omega\, x \neq_{\lambda l} \Omega$, da $\lambda x.\Omega\, x$ eine WHNF ist.

Wir wollen uns nun davon überzeugen, dass es im „lazy“ λ-Kalkül mehr gleiche Ausdrücke gibt als im klassischen λ-Kalkül. Wir benötigen also zwei Ausdrücke s und t, für die wir zeigen können, dass $s =_{\lambda l} t$, aber $s \neq_\lambda t$. Die Extensionalität kann man im klassischen λ-Kalkül zwar nicht folgern, aber ebenso wenig kann man für diese Fälle eine Ungleichheit zeigen. Wir machen uns also eine andere Eigenschaft des „lazy“ λ-Kalküls zu Nutze: Das Vorhandensein größter Elemente – es gibt nämlich mehrere. So stellt auch $(Y2\,K)$, mit $Y2 \equiv (\lambda f.\,(\lambda x\,y\,.f\,(y\,x\,x))\;(\lambda x\,y\,.f\,(y\,x\,x))\;(\lambda x\,y\,.f\,(y\,x\,x)))$ ein größtes Element dar (Beweis: siehe [Man99, Seite 15]) und somit gilt $Y\,K =_{\lambda l} Y2\,K$. Man kann aber auch feststellen, dass Y und $Y2$ nicht zu einem gemeinsamen Ausdruck reduziert werden können (siehe [Man99]) und somit aufgrund der Church-Rosser-Eigenschaft $Y \neq_\lambda Y2$ gelten muss, und damit auch $Y\,K \neq_\lambda Y2\,K$. Es gilt also: $\lambda \subset \lambda l$.

3.3.4 Call-by-Value Reduktion

Will man die Reduktion statt in Normalordnung in Anwendungsordnung betrachten, so muss man dafür sorgen, dass Redexe ausschließlich Werte enthalten dürfen, so dass nur für Terme der Form $r \equiv (\lambda x.s)\;(\lambda y.t)$ eine b-Kontraktion durchgeführt wird. Dazu verbieten wir die b-Kontraktion für Redexe der Form $r \equiv (\lambda x.s)\;t$, in denen t kein Wert ist und erweitern den Reduktionskontext so, dass t zu einem Wert reduziert werden kann.

Eine b-Kontraktion im Call-by-Value λ-Kalkül ist also definiert durch

$$(\lambda x.\, s)\; t \underset{b_v}{\rightarrow} s\{t / x\},\ t \text{ ist eine WHNF}$$

Reduktionskontexte sind nun:

$$R_v ::= [\,] \mid (\lambda x.\, t)\; R_v \mid R_v\; t$$

Die Reduktionsrelation ist somit gegeben durch $R_v(s) \underset{b_v\,,\,R_v}{\rightarrow} R_v(t)$ gdw. $s \underset{b_v}{\rightarrow} t$.

Als Gleichheitsdefinition behalten wir die gewohnte Verhaltensgleichheit bei; wie bereits erwähnt, verändert sich ja durch eine geänderte Auswertung nur die Beobachtung und nicht der Gleichheitsbegriff an sich. Bezüglich der Auswertung ändert sich nun, dass Programme, die eine WHNF haben, nicht immer terminieren – analog zu Abschnitt 2.1.3 betrifft dies z.B. den Ausdruck $K\ I\ \Omega$, denn

$$K\ I\ \Omega \underset{b_v\,,\,R_v}{\rightarrow} (\lambda y.I)\ \Omega \underset{b_v\,,\,R_v}{\rightarrow}^{*} (\lambda y.I)\ \Omega,$$

während in Normalordnung wie folgt reduziert wird:

$$K\ I\ \Omega \underset{b\,,\,R}{\rightarrow} (\lambda y.I)\ \Omega \underset{b\,,\,R}{\rightarrow} I.$$

Im Call-by-Value λ-Kalkül gilt also $K\ I\ \Omega = \Omega$, während im „lazy“ λ-Kalkül, den wir im Unterschied zum Call-by-Value λ-Kalkül auch als Call-by-Name λ-Kalkül bezeichnen wollen, $K\ I\ \Omega \neq \Omega$ gilt. Die Gleichheiten der beiden Kalküle sind also durchaus verschieden.

4 Funktionale Kernsprachen mit algebraischen Datentypen und weitere Kalküle

Im letzten Kapitel wurde ja bereits erwähnt, dass es sich bei den meisten funktionalen Programmiersprachen um einen syntaktisch angereicherten Lambda-Kalkül mit Datentypen handelt. Im Folgenden wollen wir einen Blick auf verschiedene Kernsprachen werfen, die ebenfalls syntaktisch schlicht gehalten sind, aber alle grundlegenden Aspekte einer funktionalen Programmiersprache enthalten. Der Begriff der Kernsprache entstammt der Idee, dass man die Übersetzung eines Programms, z.B. in Maschinensprache, aufteilen kann in eine Übersetzung mittels Precompiler in die Kernsprache und eine anschließende Übersetzung der Kernsprache in Maschinensprache durch den Compiler.

Wir wollen uns zunächst mit der funktionalen Kernsprache KFP (Kernsprache zur Vorlesung „Funktionale Programmierung“) aus [Sch00] beschäftigen. Anhand der Erweiterung KFP+ werden wir dann auch sehen, welcher Zusammenhang zwischen dem Lambda-Kalkül und Funktionsdefinitionen, wie in Kapitel 2 beschrieben, besteht. Danach folgt eine kurze Vorstellung der Sprache PCF (Programming Language for Computable Functions) aus [Plo77], bei der Rekursionen mittels eines eigenen Fixpunkt-Operators dargestellt werden. Zu guter Letzt betrachten wir anhand des Call-by-Need Lambda-Kalküls, mit welchen Auswertungsregeln Sharing explizit dargestellt werden kann.

4.1 Die Kernsprache KFP

Die funktionale Kernsprache KFP aus [Sch00] ist eine Erweiterung des λ-Kalküls und enthält neben algebraischen Datentypen ein Case-Konstrukt, das mit einer eigenen Reduktionsregel einher kommt.

Betrachten wir zunächst die Syntax der Kernsprache KFP:

Datentypen werden analog zu Abschnitt 2.3.2 durch Angabe ihrer Konstruktoren definiert. Wir wollen zwischen der Angabe von Typen und der Definition von Ausdrücken unterscheiden und sehen deshalb für unsere Definition der Sprache die Datentypen als gegeben an, z.B. durch eine Auflistung der Typen zusammen mit den zugehörigen Konstruktoren und deren Stelligkeit, etwa so:

```
Bool
     True 0
     True 0

List
     Nil 0
     Cons 2
```

Für einen gegebenen Konstruktor c wissen wir also immer dessen Typ und dessen Stelligkeit $ar(c)$.

Wir können nun die gültigen Ausdrücke definieren – diese bestehen aus Variablen, Funktionen und Anwendungen wie im λ-Kalkül, sowie Konstruktoren und dem eben erwähnten Case-Konstrukt:

Definition 4.1.1. Die Sprache der KFP-Ausdrücke wird definiert durch:

EXP ::= *Kons* | *V* | λ*V*.*EXP* | (*EXP EXP*) |
(`case`$_{Typ}$ *EXP* (*Pat* $_1 \rightarrow Exp_1$) ... (*Pat*$_n \rightarrow Exp_n$))

Hierbei muss *n* im Case-Ausdruck die Anzahl der Konstruktoren von *Typ* sein und für jeden Konstruktor von *Typ* muss es genau ein Pattern geben. Das Konstrukt *Pat*$_i \rightarrow Exp_i$ heißt case-*Alternative*.

Kons ::= *Konstruktor* $Exp_1 \ldots Exp_n$

n muss die Stelligkeit des Konstruktors sein.

V ::= $x \in Var$

Pat ::= *Konstruktor* $V_1 \ldots V_n$

Die Variablen V_i müssen verschieden sein, und *n* muss die Stelligkeit des Konstruktors sein. Diese Konstrukte nennt man *Pattern* (*Muster*)

Namenskonvention: Mit den Buchstaben *r*, *s*, *t* bezeichnen wir KFP-Terme, mit *x*, *y*, *z* Variablen aus *Var*, *c* oder *k* benutzen wir für Konstruktoren und *p* für Pattern.

Die für den λ-Kalkül beschriebenen Klammerkonventionen behalten wir bei und lassen auch beim Case-Ausdruck ggf. die äußeren Klammern weg. Den Case-Ausdruck nennen wir einfach *Fallunterscheidung*. Die gewohnte Fallunterscheidung für Bool'sche Ausdrücke

`if` *r* `then` *s* `else` *t*

kann man darstellen durch den Ausdruck

`case`$_{\texttt{Bool}}$ *r* (`True` $\rightarrow$ *s*) (`False` $\rightarrow$ *t*).

Im Bezug auf gebundene Variablen wollen wir zusätzlich zu der bisherigen Definition der Bindung durch λ-Ausdrücke bzw. Abstraktionen auch eine Bindung durch Pattern einführen:

Definition 4.1.2. Die Bindungsregeln in KFP sind:

- Im Ausdruck $\lambda x.\ t$ ist x eine gebundene Variable, der Gültigkeitsbereich (Skopus) ist t.

- In der Case-Alternative $((c\ x_1\ ...\ x_n) \rightarrow t)$ werden die Variablen x_i durch das Pattern gebunden, der Gültigkeitsbereich (Skopus) ist t.

Aufgrund dieser Bindungsregeln ermöglichen Case-Ausdrücke ein vollständiges Pattern-Matching wie in Abschnitt 2.3.3 beschrieben. Die Wirkungsweise wird bei Betrachtung der Case-Reduktionsregel im nächsten Abschnitt deutlich.

4.1.1 Reduktionsregeln

Wir wollen nun die Auswertung durch Angabe der Reduktionsregeln beschreiben. Dazu müssen wir zunächst die Definition der Werte um Konstruktoren erweitern:

Definition 4.1.3. Ein Wert bzw. eine WHNF in KFP ist entweder eine Abstraktion $\lambda x.\ t$ oder ein Ausdruck der Form $c\ t_1\ ...\ t_n$, wobei c ein Konstruktor ist und $n \leq ar(c)$.

Wir unterscheiden die Werte nach ihrer Struktur als

- FWHNF (Funktions-WHNF), wenn die WHNF eine Abstraktion ist oder ein Ausdruck der Form $c\ t_1\ ...\ t_n$, mit $n < ar(c)$, oder

- SCWHNF (gesättigte Konstruktor-WHNF[24]), wenn die WHNF ein Ausdruck der Form $c\ t_1\ ...\ t_n$, mit $n = ar(c)$ ist.

Widmen wir uns den Reduktionsregeln:

Definition 4.1.4. Die *unmittelbare Reduktion* eines Ausdrucks ist nach den beiden folgenden Regeln erlaubt:

Beta: $((\lambda x.s)\ t) \underset{Beta}{\rightarrow} s\{t / x\}$

Case: $(\texttt{case}_{Typ}\ (c\ t_1\ \ldots\ t_n)\ \ldots\ (c\ x_1\ \ldots x_n \rightarrow s)\ \ldots) \underset{Case}{\rightarrow} s\{t_1 / x_1, \ldots, t_n / x_n\}$

Ein *Redex* ist nun ein Ausdruck, der unmittelbar reduziert werden kann.

Definition 4.1.5. *Reduktionskontexte* sind:

$R ::= [] \mid (R\ t) \mid \texttt{case}_{Typ}\ R\ (p_1 \rightarrow t_1)\ \ldots\ (p_n \rightarrow t_n)$

Wir können also unsere Reduktionsrelation auf gewohnte Weise mit Hilfe unmittelbarer Reduktion und Reduktionskontexten angeben:

[24] engl. saturated constructor-WHNF

Definition 4.1.6. Wir definieren den Normalordnungs-Reduktionsschritt als

$$R[s] \underset{KFP,\,R}{\rightarrow} R[t] \text{ gdw. } s \underset{Beta}{\rightarrow} t \vee s \underset{Case}{\rightarrow} t.$$

Für $R[s]$ stellt der Unterterm s (zusammen mit seiner Position) einen *Normalordnungsredex* dar. Als Normalordnungsreduktion bezeichnen wir die reflexiv-transitive Hülle $\underset{KFP,\,R}{\rightarrow^*}$.

Wie gewohnt ergibt sich die Auswertung als Normalordnungsreduktion zu einem Wert:

Definition 4.1.7. In KFP ist die Auswertungsrelation $\downarrow$ gegeben durch:

$$s \downarrow t \text{ gdw. } s \underset{KFP,\,R}{\rightarrow^*} t \text{ und } t \text{ ist eine WHNF.}$$

Anmerkung: Wir verzichten hier auf einen Vergleich mit einer Darstellung als „big step“ Semantik und benutzen im Folgenden in allen Fällen, in denen wir nicht zwischen „big step“ und „small step“ Semantiken unterscheiden, auch das Symbol $\Downarrow$ für die Relation $\downarrow$, um eine einheitliche Notation für den Umgang mit der kontextuellen Gleichheit benutzen zu können. Insbesondere schreiben wir wie gewohnt $s \Downarrow$, falls die Auswertung von s terminiert und $t \Uparrow$, falls sie divergiert.

Nicht immer kann ein Normalordnungs-Reduktionsschritt für einen Term, der keine WHNF ist, ausgeführt werden: So kann z.B. $\texttt{case}_{\texttt{Bool}}\ \texttt{Nil}\ (\texttt{True} \rightarrow t_1)\ (\texttt{False} \rightarrow t_2)$ nicht weiter reduziert werden, da `Nil` nicht zum Typ `Bool` gehört. Dies kann man als Typfehler zur Laufzeit ansehen. Wir betrachten das Auftreten solcher Fehler der Einfachheit halber als Nichtterminierung. Weitere Betrachtungen zu Typen und Typregeln findet man in [Sch00].

Wir wollen unseren gewohnten Begriff der kontextuellen Approximation bzw. Gleichheit im Sinne der Beobachtung der Terminierung bei Einsetzung in Programmkontexte beibehalten, dazu müssen wir noch die Programmkontexte in KFP definieren:

Definition 4.1.8. *Programmkontexte* sind:

$$\begin{aligned} C ::= \ & [] \mid (t\ C) \mid (C\ t) \mid \lambda x.C \mid \texttt{case}_{Typ}\ C\ (p_1 \rightarrow t_1) \ldots (p_n \rightarrow t_n) \mid \\ & \texttt{case}_{Typ}\ t\ (p_1 \rightarrow t_1) \ldots (p_1 \rightarrow C) \ldots (p_n \rightarrow t_n) \mid \\ & k\ t_1 \ldots C \ldots t_n, \text{ für alle Konstruktoren } k \end{aligned}$$

Nun können wir die gewohnten Definitionen

$$s \leq_c t \Leftrightarrow \forall\ C[]: C[s] \Downarrow \Rightarrow C[t] \Downarrow$$

und

$$s =_c t \Leftrightarrow s \leq_c t \wedge t \leq_c s$$

der kontextuellen Approximation und der kontextuellen Gleichheit verwenden.

4.1.2 Rekursive Superkombinatoren: KFP+

Die Erweiterung von KFP zu KFP+ erlaubt die Angabe von Funktionsdefinitionen im gewohnten funktionalen Stil und somit auch rekursives Programmieren. Die Funktionsnamen (Superkombinatornamen) werden innerhalb von Ausdrücken als Konstanten angesehen und die Reduktion erfolgt durch eine spezielle Regel für Superkombinatoren. Die Sprache KFP+ ähnelt der Kernsprache aus [PJL91], wobei diese aber noch ein `let`- und ein `letrec`-Konstrukt für lokale Definitionen beinhaltet und alle Konstruktoren auf einen zweistelligen `Pack`-Konstruktor zurückgeführt werden.

Definition 4.1.9. Syntax von KFP+:

Die Syntax der KFP+-Ausdrücke ist wie KFP, nur um Superkombinatornamen erweitert.

Es gibt zusätzlich eine Menge von Funktionsdefinitionen:

DEF ::= *Funktionsname* $V_1 \ldots V_n = EXP$.

Folgende Bedingungen müssen eingehalten werden:

- Die Variablen $V_1 \ldots V_n$ sind voneinander verschieden und nur diese Variablen kommen im Ausdruck *EXP* frei vor.
- Die Namensräume für Variablen, Konstruktoren und Funktionsnamen sind disjunkt.
- Jede Funktion wird höchstens einmal definiert.

Die Stelligkeit von Funktionsnamen *f*, geschrieben als *ar*(*f*), ist die Anzahl der Variablen in der Definition von *f*.

Definition 4.1.10. Ein KFP+-*Programm* besteht aus

1. einer Menge von Typen und Konstruktoren wie in KFP,
2. einer Menge von Funktionsdefinitionen
3. und aus einem Ausdruck. Dieser kann auch nach der main-Konvention als

   main = *Exp*

   definiert werden.

Kontexte und Reduktionskontexte in KFP+ sind analog zu KFP definiert, nur erweitert um Superkombinatornamen in Ausdrücken. Zur Auswertung des Programms (d.h. des main-Ausdrucks) steht zusätzlich zur Beta- und Case-Regel noch die SK-Beta-Reduktionsregel zur Verfügung:

Definition 4.1.11. Anwendungen von Superkombinatoren auf ihre Argumente können wie folgt unmittelbar reduziert werden:

$$(f\ a_1 \ldots a_n) \underset{SK-Beta}{\rightarrow} r\{a_1/x_1, \ldots, a_n/x_n\},$$

wenn f definiert ist durch: $f\ x_1 \ldots x_n = r$.

Als Auswertung dient nun die gewohnte Normalordnungsreduktion (allerdings inklusive SK-Beta-Regel) zur WHNF. Allerdings wollen wir auch diesen Wert-Begriff erweitern, so dass auch ungesättigte Funktionsanwendungen einen Wert darstellen:

Definition 4.1.12. Eine KFP+-WHNF ist entweder

- ein Ausdruck $(c\ t_1 \ldots t_n)$, wobei $n \leq ar(c)$; oder
- ein Ausdruck $(f\ t_1 \ldots t_n)$, wobei $n < ar(c)$; oder
- ein Ausdruck $\lambda x.\, t$

Damit ist also die Auswertung für KFP+ festgelegt.

4.1.3 Zusammenhang zwischen KFP und Superkombinatoren

Wir wollen uns jetzt den Zusammenhang zwischen KFP und KFP+ näher anschauen. Dazu betrachten wir, wie man Rekursionen auflösen kann, um entsprechende Berechnungen auch in KFP zu ermöglichen. Der einfachste Fall einer rekursiven Funktionsdefinition ist, wenn der Funktionsrumpf als einzigen Superkombinator den eigenen Funktionsnamen enthält, die Rekursion also nicht verschränkt sein kann, indem sich mehrere Funktionen gegenseitig aufrufen. Sei $f\ x = r$ eine rekursive Funktionsdefinition, so dass r den Funktionsnamen f enthält, aber keine anderen Superkombinatoren:

Eine äquivalente Definition (im Sinne von $=_c$) ist $f' = \lambda x.\, r'$, mit $r' \equiv r\{f' / f\}$.

Zudem gilt: $f' =_c (\lambda z\, x.\, s)\, f'$, mit $s \equiv r'\{z / f'\}$. Somit ist f' ein Fixpunkt von $F' \equiv \lambda z\, x.\, s$, d.h. $f' =_c F'\ f'$. Wir können f' also nicht-rekursiv anhand eines Fixpunktkombinators Y definieren, für den gilt: $Y\ t =_c t\ (Y\ t)$, für alle Terme t. Dies leistet

$$Y \equiv \lambda x.(\lambda y.\, x\ (y\ y))\ (\lambda y.\, x\ (y\ y))$$

aus Definition 3.1.7. , denn:

$$(Y\ t) \rightarrow (\lambda y.\, t\ (y\ y))\ (\lambda y.\, t\ (y\ y)) \rightarrow t\ ((\lambda y.\, t\ (y\ y))\ (\lambda y.\, t\ (y\ y))) =_c t\ (Y\ t).$$

Wir definieren also unsere Funktion auf nicht-rekursive Weise wie folgt:

$$F \equiv Y\ F'$$

f und F verhalten sich nun gleich:

$$f\ t \rightarrow r\{t / x\}$$

$$F\ t \equiv (Y\ F')\ t =_c F'\ (Y\ F')\ t \equiv F'\ F\ t \to^2 s\{F / z, t / x\} \equiv r\{F / f, t / x\}.$$

Für den Fall, dass die Rekursion verschränkt ist über die Superkombinatoren S_1, ..., S_n, kann eine einzige (einfach-rekursive) Funktion F definiert werden, die anhand einer Fallunterscheidung des ersten (ganzzahligen) Argumentes für die Anwendung ($F\ i$) gerade S_i implementiert. Somit müssen nur die Vorkommen aller S_i durch $F\ i$ ersetzt werden. Die Problematik verschiedener Stelligkeiten kann dabei durch Hinzufügen von Dummy-Argumenten umgangen werden. Für die Details einer Übersetzung von KFP+ nach KFP siehe [Sch00, Abschnitt 7]. Zu beachten ist, dass bei einer Übersetzung einer Funktion mit mehreren Argumenten in eine Lambda-Abstraktion die Superkombinator-Anwendung mittels SK-Beta-Regel nur dann reduziert werden kann, wenn genügend Argumente zur Verfügung stehen, während in KFP ein Argument zur Beta-Reduktion ausreicht. Dies ändert aber nichts am Terminierungsverhalten, da ja ungesättigte Superkombinator-Anwendungen einen Wert darstellen; es bleibt sogar die kontextuelle Ordnung erhalten, siehe [Sch00, Satz 7.17.].

Wir können also die gewohnten Funktionsdefinitionen durch äquivalente Lambda-Abstraktionen darstellen. Wir wollen uns jetzt noch davon überzeugen, dass auch alle Lambda-Abstraktionen mit Superkombinatoren dargestellt werden können. Dazu müssen lokale Abstraktionen, die lokal freie Variablen enthalten, eliminiert werden. Dies ist anhand der Transformation „Lambda Lifting“ möglich:

Definition 4.1.13. Wir definieren die *Lambda-Lifting Transformation* wie folgt:

$$\lambda x.\, t \underset{ll}{\to} (\lambda z.\, \lambda x.\, t\{z / y\})\ y, \text{ wobei } y \in FV(t) \text{ und } z \text{ ist eine neue Variable.}$$

Lambda-Lifting ist also eine spezielle b-Expansion, die eine freie Variable aus einer Abstraktion auf die rechte Seite einer Applikation „liftet“. Durch Anwendung der Lambda-Lifting Transformation auf lokale Abstraktionen können wir sicherstellen, dass diese keine freien Variablen innerhalb der Abstraktion enthalten. (Dies ist stets mit endlich vielen Lambda-Lifting Transformationen möglich, siehe [Sch00].) Einen geschlossenen Ausdruck bzw. Kombinator, bei dem alle lokalen Abstraktionen auf diese Weise geliftetet wurden, nennen wir in diesem Zusammenhang auch Superkombinator, denn wir können nun alle enthaltenen Abstraktionen durch Superkombinatornamen ersetzen, indem wir für jede Abstraktion $\lambda x_1 ... x_n\,.\ t$ eine Funktionsdefinition $Name\ x_1\ ...\ x_n = t$ einführen.

4.2 PCF: Programming Language for Computable Functions

Die Sprache PCF aus [Plo77] basiert auf einem logischem Kalkül von Dana Scott [Sco69/93], bekannt als „logic of computable functions“ bzw. LCF, und dient grundlegenden Untersuchungen funktionaler Sprachen. Neben λ-Abstraktionen enthält PCF Konstrukte mit einem Fixpunkt-Operator μ. Die Basisdatentypen sind natürliche Zahlen und Bool’sche Wahrheitswerte. Neben der Fallunterscheidung `if then else` sind die Funktionen `pred`, `succ` und `zero?` vordefiniert, mit denen der Vorgänger bzw. Nachfolger einer natürlichen Zahl erhalten werden kann, oder geprüft wird, ob eine Zahl Null ist. Üblicherweise wird die Sprache PCF mit einem einfachen (monomorphen) Typsystem versehen, d.h. alle Ausdrücke haben einen Typ, der keine (Typ-)Variablen

enthält, und dieser Typ ist eindeutig. Wir halten uns im Folgenden an die Definitionen aus [Sch01]:

Definition 4.2.1. Syntax von PCF:

Typen:

$$t ::= \texttt{num} \mid \texttt{Bool} \mid t \to t$$

Ausdrücke:

$$\begin{aligned} E ::= \; & \texttt{True} \mid \texttt{False} \mid \\ & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \\ & \texttt{pred}\ E \mid \texttt{succ}\ E \mid \\ & \texttt{zero?}\ E \mid \\ & (\texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E) \mid \\ & V \mid \lambda V :: t.E \mid (E\ E) \mid \mu V :: t.E \end{aligned}$$

$$V ::= \; x \in Var$$

In den λ- und μ- Konstrukten müssen die Variablen mit einem (monomorphen) Typ versehen sein, damit man den Typ der Ausdrücke angeben kann. Gültige Ausdrücke müssen wohlgetypt sein, d.h. es muss ein Typ hergeleitet werden können. Die vordefinierten Konstanten und Funktionen haben dabei folgende Typen:

`True`, `False`	$:: \texttt{Bool}$
`0`, `1`, `2`, …	$:: \texttt{num}$
`pred`, `succ`	$:: \texttt{num} \to \texttt{num}$
`zero?`	$:: \texttt{num} \to \texttt{Bool}$
`if-then-else`	$:: \texttt{Bool} \to a \to a \to a$, für alle a

`if-then-else` kann man als generisches Konstrukt ansehen, das für jeden Typ t zu einem speziellen $\texttt{if-then-else}_t$ instanziiert wird.

Die Typen für λ-Abstraktionen und μ-Konstrukte, sowie für Applikationen können nach den folgenden Typregeln hergeleitet werden:

$$\frac{t :: t_2}{(l\,x :: t_1 . t) :: t_1 \to t_2}, \qquad \frac{(l\,x :: t\,.t) :: t \to t}{(m x :: t\,.\,t) :: t}, \qquad \frac{s :: t_1 \to t_2,\ t :: t_1}{(s\ t) :: t_2}$$

Wir wollen das Typsystem ab jetzt weitgehend vernachlässigen. Zu beachten ist allerdings, dass das Typsystem die Verwendung der λ-Ausdrücke stark einschränkt. So hat z.B. die Selbstapplikation $(x\ x)$ keinen Typ, da es keinen unendlichen Typ $t = t_1 \to t_2$, mit $t_1 = t$, gibt. Insbesondere gibt es keine Möglichkeit, einen Fixpunkt-Kombinator wie Y zu

definieren, Rekursionen können somit ausschließlich mittels des μ-Operators dargestellt werden (siehe [Sch01, Korollar 5.6]).

4.2.1 Operationale Semantik von PCF

Wir definieren die Auswertung von PCF wie gewohnt mittels Reduktionsregeln und Reduktionskontexten. Zunächst müssen wir natürlich unseren Wertebegriff anpassen und wollen dabei neben Abstraktionen die Konstanten der Basisdatentypen mit aufnehmen:

Definition 4.2.2. Werte (WHNFs) in PCF sind Lambda-Abstraktionen, `True`, `False` und die natürlichen Zahlen.

Jetzt können wir die Reduktion angeben:

Definition 4.2.3. Sei n eine natürliche Zahl, dann können wir die Reduktionsregeln wie folgt definieren:

`if True then` s `else` t	$\rightarrow s$
`if False then` s `else` t	$\rightarrow t$
`pred` n	$\rightarrow n$ - 1, falls $n > 0$
`pred 0`	$\rightarrow$ `0`
`succ` n	$\rightarrow n + 1$
`zero?` n	$\rightarrow$ `False`, falls $n > 0$
`zero? 0`	$\rightarrow$ `True`
$(\lambda x.s)\ t$	$\rightarrow s\{t / x\}$
$(\mu x.t)$	$\rightarrow (t\{(\mu x.t) / x\})$

Reduktionskontexte sind:

$R ::= [] \mid (R\ t) \mid$ `if` R `then` s `else` $t \mid$ `pred` $R \mid$ `succ` $R \mid$ `zero?` R

Die Reduktionsrelation ist also definiert durch: $R[s] \underset{R}{\rightarrow} R[t]$ gdw. $s \rightarrow t$

Somit ist die Auswertung definiert durch: $M \downarrow N$, gdw. $M \underset{R}{\overset{*}{\rightarrow}} N$ und N ist eine WHNF.

4.3 Call-by-need Lambda-Kalkül

Ein Call-by-need Lambda-Kalkül λ_{Need} dient zur formalen Beschreibung der nichtstrikten Auswertung mit Sharing, bzw. lazy-Auswertung, wie in Abschnitt 2.1.3 beschrieben. Grundlegende Untersuchungen zu Call-by-need Lambda-Kalkülen findet man in [AF97], [AFM+95] und [MOW98], wir beziehen uns im Folgenden auf [MOW98]. λ_{Need} enthält dort neben den gewohnten Abstraktionen und Applikationen der Sprache Λ noch ein let-Konstrukt, das die gemeinsame Benutzung von Funktionsparameter-Ausdrücken im Funktionsrumpf darstellt, so dass ein unnötiges Kopieren wie im „lazy“ Lambda-Kalkül λ_l

verhindert wird. Das let-Konstrukt kann dabei auch mittels Applikationen in der Sprache Λ codiert werden, es sind lediglich spezielle Auswertungsregeln nötig.

Definition 4.3.1. Die Sprache Λ_{Need} wird definiert durch die kontextfreie Grammatik mit der Produktionenmenge

$$EXP ::= V \mid \lambda V.EXP \mid EXP\ EXP \mid \texttt{let}\ V = EXP\ \texttt{in}\ EXP$$

$$V ::= \quad x \in Var$$

Die Reduktionsregeln unterscheiden sich nun natürlich stark vom „lazy“ Lambda-Kalkül. Zur besseren Unterscheidung von den Werten des „lazy“ Lambda-Kalküls bezeichnen wir die Ergebnisse einer Auswertung im Call-by-Need Lambda-Kalkül als *Antworten*[25], denn bei diesen erlauben wir auch eine let-Hülle:

Definition 4.3.2. Die Menge der Antworten ist definiert durch:

$$A ::= \lambda x.t \mid \texttt{let}\ x = t\ \texttt{in}\ A$$

Mit Werten bzw. WHNFs meinen wir also nach wie vor Abstraktionen.

Definition 4.3.3. Die folgenden Reduktionsschritte sind in λ_{Need} erlaubt:

(*I*)	$(\lambda x.s)\ t$	$\rightarrow \texttt{let}\ x = t\ \texttt{in}\ s$
(*V*)	$\texttt{let}\ x = v\ \texttt{in}\ C[x]$ v ist ein Wert	$\rightarrow \texttt{let}\ x = v\ \texttt{in}\ C[v]$
(*C*)	$(\texttt{let}\ x = r\ \texttt{in}\ s)\ t$	$\rightarrow \texttt{let}\ x = r\ \texttt{in}\ s\ t$
(*A*)	$\texttt{let}\ y = (\texttt{let}\ x = r\ \texttt{in}\ s)\ \texttt{in}\ t$	$\rightarrow \texttt{let}\ x = r\ \texttt{in let}\ y = s\ \texttt{in}\ t$
(*G*)	$\texttt{let}\ x = s\ \texttt{in}\ t$	$\rightarrow t$, falls $x \notin FV(t)$

Programmkontexte sind dabei:

$$C ::= [] \mid \lambda x.C \mid C\ t \mid t\ C \mid \texttt{let}\ x = C\ \texttt{in}\ t \mid \texttt{let}\ x = t\ \texttt{in}\ C$$

[25] engl. *Answers*

Regel (*I*), „Introduction“, erzeugt aus einer Anwendung eine let-Bindung. Die let-Bindung repräsentiert die Referenz auf das Argument N anstelle der Funktionsvariable x im instanziierten Funktionsrumpf M.

Regel (*V*), „Value“, ersetzt eine let-gebundene Variable durch einen Wert (Dereferenzieren). Da nur Werte kopiert werden, besteht keine Gefahr, dass man noch zu reduzierende Ausdrücke dupliziert.

Regel (*C*), „Commute“, erlaubt es let-Bindungen mit Anwendungen zu kommutieren, so dass eine let-Bindung aus dem Funktionsteil einer Anwendung herausgezogen wird (auch „lift“ genannt).

Regel (*A*), „Associate“ überführt links-stehende in rechts-stehende let-Ausdrücke.

Regel (*G*), „Garbage Collection“, entfernt eine let-Bindung, deren Variable x nicht mehr im Term N auftaucht.

Die Relation $\underset{C}{\rightarrow}^*$ mit $C[s] \underset{C}{\rightarrow} C[t]$ gdw. $s \rightarrow t$ entspricht der *b*-Reduktion mit Sharing:

Resultat: Die Sprache Λ_{Need} kann nach Λ übersetzt werden, so dass für die Übersetzung s gilt: $s \underset{C}{\rightarrow}^* t \Rightarrow s(s) \underset{b,C}{\rightarrow}^* s(t)$. Zudem gilt $s(s) \underset{b,C}{\rightarrow}^* t_0 \Rightarrow \exists\, t\colon s \underset{C}{\rightarrow}^* t \wedge t_0 \underset{b,C}{\rightarrow}^* s(t)$. (vgl. [MOW98, Proposition 28])

4.3.1 Normalordnungsreduktion in $\mathbf{l}_{\text{Need}}$

Definition 4.3.4. Reduktionskontexte sind:

$$R, R_i ::= [] \mid R\ t \mid \texttt{let}\ x = t\ \texttt{in}\ R \mid \texttt{let}\ x = R_0\ \texttt{in}\ R_1[x]$$

Um eine eindeutige Auswertungsreihenfolge zu erhalten, reicht es jedoch nicht aus, die Reduktionsschritte ausschließlich in Reduktionskontexten auszuführen. So hätte z.B. der Ausdruck $\texttt{let}\ x = v_0\ \texttt{in}\ \texttt{let}\ y = v_1\ \texttt{in}\ x\ y$ zwei Normalordnungsredexe, da wir den gesamten Ausdruck im leeren Kontext reduzieren könnten oder aber den Unterausdruck $\texttt{let}\ y = v_1\ \texttt{in}\ x\ y$ im Reduktionskontext $\texttt{let}\ x = v_0\ \texttt{in}\ []$. Wir müssen also die Regeln leicht ändern:

Definition 4.3.5. Normalordnungs-Reduktionsregeln:

(I_{no}) $(\lambda x.s)\ t \rightarrow \texttt{let}\ x = t\ \texttt{in}\ s$

(V_{no}) $\texttt{let}\ x = v\ \texttt{in}\ R[x] \rightarrow \texttt{let}\ x = v\ \texttt{in}\ R[v]$

(C_{no}) $(\texttt{let}\ x = s\ \texttt{in}\ a)\ t \rightarrow \texttt{let}\ x = s\ \texttt{in}\ a\ t$

(A_{no}) $\texttt{let}\ y = (\texttt{let}\ x = s\ \texttt{in}\ a)\ \texttt{in}\ R[y] \rightarrow \texttt{let}\ x = s\ \texttt{in}\ \texttt{let}\ y = a\ \texttt{in}\ R[y]$

(v ist ein Wert, a ist eine Antwort.)

Für die Anwendung einer beliebigen Normalordnungs-Reduktionsregel benutzen wir das Symbol $\underset{no}{\rightarrow}$. Die Reduktionsrelation ergibt sich somit als $R[s] \underset{no,R}{\rightarrow} R[t]$ gdw. $s \underset{no}{\rightarrow} t$.

Definition 4.3.6. Die Auswertungsrelation für λ_{Need} ist definiert durch:

$t \downarrow a$ gdw. $t \underset{no,R}{\overset{*}{\rightarrow}} a$ und a ist eine Antwort.

Resultat: Die Normalordnungsreduktion hat die gewünschten Eigenschaften (vgl. [MOW98, Theorem 10]):

- Sie ist eindeutig: Ein Term ist entweder eine Antwort oder hat einen einzigen Normalordnungsredex.
- Sie ist gültig bzgl. $\underset{C}{\overset{*}{\rightarrow}}$, d.h. es gilt: Falls $t \underset{no,R}{\overset{*}{\rightarrow}} a$, dann auch $t \underset{C}{\overset{*}{\rightarrow}} a$, für alle $t \in \Lambda_{\text{Need}}$, $a \in A$.
- Sie terminiert, falls es eine terminierende Reduktionsfolge gibt, d.h. für alle $t \in \Lambda_{\text{Need}}$, $a \in A$: Falls $t \underset{C}{\overset{*}{\rightarrow}} a$, dann gibt es eine Antwort a_0, so dass $t \underset{no,R}{\overset{*}{\rightarrow}} a_0$. (Mit der Gültigkeit der Normalreduktion, sowie Proposition 28 und Theorem 32 aus [MOW98] folgt auch $a_0 \underset{C}{\overset{*}{\rightarrow}} a$.)

Die Auswertung von Programmen aus $\Lambda \subset \Lambda_{\text{Need}}$ entspricht nun in λ_{Need} der Auswertung in λ_l, bis auf die Anzahl der Reduktionsschritte. Insbesondere gilt:

Resultat: Die Verhaltensgleichheiten von λ_{Need} und λ_l stimmen für Terme aus $\Lambda \subset \Lambda_{\text{Need}}$ überein, d.h. für alle M, N $\in \Lambda$:

$M =_l N$ gdw. $M =_{\text{Need}} N$.

(Siehe Theorem 32 aus [MOW98]).

Man kann auf das let-Konstrukt verzichten, indem man `let` x `=` s `in` t als syntaktisch gleich zu $(\lambda x.t)\ s$ ansieht (vergleiche Regel (*I*): Diese wird dann nicht mehr benötigt). Für die Details dazu siehe [MOW98, Abschnitt 6].

Für weitere Eigenschaften von Call-by-Need Kalkülen siehe [AF97], [AFM+95], [MOW98].

5 Termersetzungssysteme höherer Ordnung

Wir beschäftigen uns nun mit Termersetzungssystemen. Diese bestehen aus der Definition der erlaubten Ausdrücke und einer Menge einfacher Reduktionsregeln zur textuellen Ersetzung von Ausdrücken. In diesen Regeln dürfen Variablen verwendet werden: In *Termersetzungssystemen erster Ordnung*[26] dürfen diese nur Konstruktor- bzw. Zahlenwerte enthalten – diese Termersetzungssysteme können so z.B. zur Lösung mathematischer Gleichungssysteme verwendet werden. In *Termersetzungssystemen höherer Ordnung*[27] dürfen die Variablen beliebige Ausdrücke enthalten, ebenso kann Variablenbindung ausgedrückt werden. Die Termersetzungssysteme höherer Ordnung sind somit geeignet, die Reduktionsschritte des Lambda-Kalküls nachzuvollziehen. Um die speziellen Variablen innerhalb der Regeln von den Variablen in Ausdrücken zu unterscheiden, nennen wir diese Metavariablen – näheres dazu im nächsten Unterabschnitt.

Im Folgenden beziehen wir uns auf die Definition der „*kombinatorischen Reduktionssysteme*“[28] aus [KOR93]. Im Gegensatz zu den Termersetzungssystemen höherer Ordnung aus [Nip93] und [MN98] kommen diese ohne einen einfach getypten Lambda-Kalkül als Metasprache aus und definieren stattdessen eigene ungetypte Abstraktions- und Substitutionsmechanismen. Bezüglich der Auswertung sind beide Systeme gleich mächtig, die Auswertung hat die gleichen Ergebnisse, nur dass die „kombinatorischen Reduktionssysteme“ unter Umständen mehrere Schritte benötigen, um einen Schritt des anderen Modells zu simulieren (siehe Vergleich zu den „surplus HRS“ in [KOR93, Seite 17]). Für den Speziallfall der „pattern rewrite systems“ aus [MN98] sind die Modelle sogar bezüglich einzelner Reduktionsschritte äquivalent (siehe Vergleich zu den „simple HRS“ in [KOR93, Seite 17]). Wenn wir von einem Termersetzungssystem höherer Ordnung sprechen, meinen wir also ab jetzt ein solches „kombinatorisches Reduktionssystem“.

Wir betrachten nun zunächst die Grundlagen der Termersetzungssysteme höherer Ordnung, insbesondere das Format der Reduktionsregeln. Im Anschluss daran wird ein anderes Regelformat vorgestellt, das zusätzlich die Betrachtung von Call-by-Value-Sprachen ermöglicht: das GDSOS-Format aus [San97]. (GDSOS steht dabei für *Global deterministische, strukturelle, operationale Semantik*[29]). Wie wir sehen werden, können GDSOS-Regeln durch die Reduktionsregeln eines Termersetzungssystems höherer Ordnung dargestellt werden, wenn wir spezielle Werte-Metavariablen erlauben. Danach beschäftigen wir uns noch mit *strukturierten Auswertungssystemen*[30] gemäß [How96] und zeigen, dass wir die speziellen Eigenschaften dieser Auswertungssysteme auch für Reduktionssysteme mit GDSOS-Reduktionsregeln folgern können, da wir die GDSOS-Regeln in (gemäß Auswertung) äquivalente strukturierte Auswertungsregeln übersetzen

[26] engl. *first order rewrite systems*, auch gebräuchlich: *term rewrite systems*
[27] engl. *higher order rewrite systems*
[28] engl. *combinatory reduction systems*
[29] engl. *Globally Deterministic Structural Operational Semantics*
[30] engl. *structured evaluation systems*

können. Wir runden das Kapitel mit Betrachtungen zur Darstellung der bisher vorgestellten Kernsprachen und Kalküle mittels Termersetzungssystemen höherer Ordnung ab.

5.1 Grundlagen

Die Definition der Ausdrücke in einem Termersetzungssystem höherer Ordnung erfolgt anhand des Begriffes einer Signatur:

Definition 5.1.1. Eine *Signatur* ist ein Paar (O, a) mit folgenden Eigenschaften:

- O ist eine Menge
- $a \in O \rightarrow \{(k_1, ..., k_n) \mid n, k_i \in \mathbb{N}_0\}$

Die Elemente von O nennen wir *Operatoren*. Für ein $F \in O$, stellt $a(F)$ die *Stelligkeit* von F dar und spezifiziert die Anzahl und die Bindungsstruktur der möglichen Argumente bzw. *Operanden* eines Operators. Die Operanden können Variablen binden: Der i-te Operand soll von der Form $x_1..x_{k_i}.t$ [31] sein, was bedeuten soll, dass die freien Vorkommen von $x_1...x_{k_i}$ im Term t gebunden sind. Wir schreiben künftig auch abkürzend $\vec{x}_i$ für $x_1...x_{k_i}$. a ist also eine Funktion, die jedem Operator aus O eine Sequenz aus natürlichen Zahlen $k_1, ..., k_n$ zuordnet, die für jeden Operanden des jeweiligen Operators die Anzahl k_i der gebundenen Variablen angibt.

Für eine gegebene Signatur ergeben sich nun die gültigen Ausdrücke wie folgt:

Definition 5.1.2. *Terme* über einer Signatur:
Sei $L = (O, a)$ eine Signatur und *Var* eine feste, abzählbar unendliche Menge von Variablen. Wir definieren die Menge $T(L)$ der Terme über L induktiv wie folgt:

$$\frac{x \in Var}{x \in T(L)},$$

$$\frac{t_1 \in T(L), ..., t_n \in T(L)}{F(\vec{x}_1. t_1, ..., \vec{x}_n. t_n) \in T(L)},$$

wobei $F \in O$, $a(t) = (k_1, ..., k_n)$ und jedes $\vec{x}_i$ eine Sequenz von k_i verschiedenen Variablen aus *Var* ist.

Wir definieren $T_0(L)$ als die Menge der geschlossenen Terme (d.h. Terme ohne freie Variablen) in $T(L)$. (Vereinfachend wollen wir künftig T_0 bzw. T schreiben, wenn L aus

[31] Wir betrachten die Variablenbindung nun losgelöst von λ-Funktionsausdrücken und werden deshalb ab jetzt Operanden, die Variablen binden, als Abstraktionen bezeichnen. Zu beachten ist, dass diese Operanden gemäß Definition 6.1.2 nicht als eigenständige Ausdrücke zulässig sind, also nur in Zusammenhang mit passenden Operatoren benutzt werden dürfen.

dem Zusammenhang eindeutig hervorgeht.) Wie gewohnt betrachten wir Terme als (syntaktisch) gleich, wenn sie bis auf Umbenennung der gebundenen Variablen gleich sind.

Beispiel 5.1.3. Wenn wir für die Darstellung der Applikation einen Operator @ benutzen, dann ist die Signatur des Lambda-Kalküls gegeben durch

$L_\Lambda=(O, a)$, wobei $O = \{\lambda, @\}$ und $a(\lambda) = (1)$, $a(@) = (0, 0)$.

Den Term $(I\ K) \equiv (\lambda x.x)\ \lambda y.\lambda z.y$ können wir darstellen als

$@(\lambda(x.\ x),\ \lambda(y.\lambda(z.y)))$.

5.1.1 Metavariablen

Wir wollen in den Reduktionsregeln Metavariablen verwenden, die Terme und Abstraktionen enthalten können. Jeder Metavariable X ordnen wir eine Stelligkeit $a(X)$ zu, die eine natürliche Zahl ≥ 0 ist und die Anzahl der gebundenen Variablen angibt. (Genau wie die k_i in der Stelligkeitssequenz eines Operators.) Metavariablen mit Stelligkeit 0 stehen für Terme, während Metavariablen mit höherer Stelligkeit für Abstraktionen stehen.

Wir definieren nun eine Erweiterung des Termbegriffes um Metavariablen:

Definition 5.1.4. *Metaterme*:
Sei *Mvar* eine feste, abzählbar unendliche Menge von Metavariablen:

Wir definieren zunächst die Mengen MT_i, wobei i die Anzahl der abstrahierten Variablen ausdrückt, wie folgt:

$\dfrac{}{x \in MT_0}$, wobei $x \in Var$,

$\dfrac{}{X \in MT_n}$, wobei $X \in Mvar$, mit $a(X) = n$,

$\dfrac{M \in MT_0}{x_1 ... x_n . M \in MT_n}$, wobei $x_i \in Var$ und die x_i sind paarweise disjunkt

$\dfrac{M_1 \in MT_0 ... M_n \in MT_0}{X\{M_1,...,M_n\} \in MT_0}$, wobei $X \in Mvar$, mit $a(X) = n$

$\dfrac{M_1 \in MT_{k_1} ... M_n \in MT_{k_n}}{F(M_1,...,M_n) \in MT_0}$, wobei $F \in O$, mit $a(F) = (k_1, ..., k_n)$

Die Metaterme sollen genau MT_0 sein.[32]

Namenskonvention: Mit den Buchstaben M, N bezeichnen wir im Folgenden Metaterme, oder auch, als Operanden benutzt, Elemente aus MT_i; des weiteren benutzen wir X, Y, Z für Metavariablen, den Buchstaben F reservieren wir für Operatoren und wie gewohnt benutzen wir r, s, t für Terme und x, y, z für Variablen.

Das Konstrukt $X\{M_1, ..., M_n\}$ beschreibt eine *Meta-Applikation*:
Die Metavariable X der Stelligkeit n steht für eine Abstraktion von n Variablen. Wenn X durch diese Abstraktion ersetzt wird, erfolgt eine Substitution der entsprechenden Variablen durch die Metaterme M_1 bis M_n: Sei s eine Instanziierung, die Metavariablen durch Abstraktionen der jeweiligen Stelligkeit ersetzt. $s(X)$ hat nun die Form $x_1...x_n . t$. Die Anwendung von s auf Metaterme entspricht normaler Substitution, außer dass die Anwendung auf Ausdrücke der Form $X\{M_1, ..., M_n\}$ wie folgt verlaufen soll:
Sei $s(X) = x_1...x_n . t$: $s(X\{M_1, ..., M_n\}) = r$, wobei $r \equiv t\{s(M_1), ..., s(M_n)/x_1, ..., x_n\}$.

5.1.2 Termersetzungsregeln

Wir können die Reduktionsregeln nun mittels Metatermen wie folgt definieren:

Definition 5.1.5. Eine Termersetzungs- bzw. Reduktionsregel eines Termersetzungssystems höherer Ordnung ist ein Paar (M, N), geschrieben als $M \to N$, so dass gilt:

1. M und N sind geschlossene Metaterme.
2. M hat die Form $F(M_1 ... M_n)$.
3. Die Metavariablen, die in N enthalten sind, sind auch in M enthalten.
4. Die Metavariablen in M treten nur in der Form $X\{x_1,...,x_k\}$ auf, wobei die x_i ($i = 1, ..., k$) Variablen sind und $a(X) = k$. Außerdem sind die x_i paarweise disjunkt.

Beispiel 5.1.6. Die b-Kontraktion kann man darstellen durch die Regel

$$@(\lambda(x.X\{x\}), Y) \to X\{Y\}, \text{ wobei } a(X) = 1 \text{ und } a(Y) = 0.$$

Wir können die Regeln bei passenden Instanzen der Metaterme anwenden. Wir müssen allerdings aufpassen, dass dadurch nicht ungewollt Variablen eingefangen werden. Konflikte lösen wir durch Umbenennungen der gebundenen Variablen innerhalb der Regeln oder der Instanzen:

Definition 5.1.7. (1) Sei $M \to N$ eine Reduktionsregel: Eine Umbenennung (der gebundenen Variablen) der Regel nennen wir eine *Variante* der Regel.

[32] Diese Definition weicht von [KOR93] ab – dort sind auch Abstraktionen Metaterme.

(2) Sei s eine Instanziierung: Eine *Variante* von s entsteht durch Umbenennung der gebundenen Variablen innerhalb einer Instanz $s(X)$.

(3) Sei $M \to N$ eine Reduktionsregel und s eine Instanziierung: Wir bezeichnen $M \to N$ als *sicher* bzgl. s, gdw. wenn es für keine Metavariable X in M und N eine Instanz $s(X)$ gibt, die eine freie Variable x enthält, falls X innerhalb einer Abstraktion steht, die x bindet.

(4) Wir bezeichnen eine Instanziierung s als sicher (in Bezug auf sich selbst), gdw. es keine zwei Instanzen $s(X)$ und $s(X')$ gibt, so dass $s(X)$ eine freie Variable x enthält, die in $s(X')$ auch gebunden vorkommt.

Da wir problemlos umbenennen können, gilt: Falls eine Regel oder Instanziierung nicht sicher ist, können wir jeder Zeit zu einer Variante wechseln, die sicher ist. Wir setzen daher im Folgenden sichere Regeln und Instanziierungen voraus.

Da wir die Regeln innerhalb von Kontexten anwenden wollen, müssen wir erst definieren, wie diese für eine gegebene Signatur aussehen sollen:

Definition 5.1.8. Sei $L = (O, a)$ eine Signatur, dann sind Kontexte gegeben durch:

$$C ::= [] \mid F(\vec{x}_1.C, ..., \vec{x}_n.t_n) \mid ... \mid F(\vec{x}_1.t_1, ..., \vec{x}_n.C), \quad \text{für alle } F \in O, \text{ mit } a(F) = (k_1, ..., k_n) \text{ und } n > 0.$$

Jetzt können wir die Anwendung einer Reduktionsregel definieren:

Definition 5.1.9. Sei $M \to N$ eine Reduktionsregel, die sicher ist für eine sichere Instanziierung s : Wir bezeichnen $s(M) \to s(N)$ als *Termersetzung* bzw. als *Kontraktion*. Der Term $s(M)$ ist ein *Redex*.

Für einen Kontext $C[]$ ist $C[s(M)] \underset{C}{\to} C[s(N)]$ ein *Termersetzungsschritt* (bzw. ein *Reduktionsschritt*). Die reflexiv-transitive Hülle $\underset{C}{\to}^*$ bezeichnen wir als *Reduktion*.

Aufgrund der Definition des Reduktionsschrittes ist es nicht nötig, explizit eine Einschränkung bzgl. der Variablen in Instanzen der Metavariablen zu formulieren, so dass unerwünschte Variablenbindung verhindert wird:

Beispiel 5.1.10. Im um Extensionalität erweiterten Lambda-Kalkül λ_h erfordert die Regel

$$(h) \quad (\lambda x.\, t\, x) \underset{h}{\to} t, \text{ wobei } x \notin FV(t),$$

die Einschränkung $x \notin FV(t)$. Eine entsprechende Reduktionsregel eines Termersetzungssystems höherer Ordnung,

$$\lambda(x.@(Z, x)) \to Z,$$

benötigt diese Einschränkung nicht, da $\rightarrow$ nicht sicher ist für s mit $s(Z) = x$ und wir somit zur Variante $\lambda(y.@(Z, y)) \rightarrow Z$ übergehen.

Definition 5.1.11. Sei R ein Termersetzungssystem höherer Ordnung mit den Termersetzungsregeln $\{R_i = M_i \rightarrow N_i \mid i \in I, I$ ist eine Indizierungsmenge$\}$:

1. R ist *nicht-überlappend*, gdw. Folgendes gilt:

   - Enthalte die linke Seite M_i von R_i die Metavariablen $Z_1\{\vec{x}_1\}$, ..., $Z_m\{\vec{x}_m\}$: Falls der R_i-Redex $s(M_i)$ einen R_j-Redex ($i \neq j$) enthält, dann muss dieser R_j-Redex bereits in einem der $s(Z_p(\vec{x}_p))$ enthalten sein,

   - ebenso, falls der R_i-Redex einen R_i-Redex enthält.

2. R ist *links-linear*, gdw. alle M_i linear sind. Ein Metaterm ist linear, gdw. es keine Mehrfachvorkommen der gleichen Metavariable gibt.

3. R ist *orthogonal*, gdw. es *nicht-überlappend* und *links-linear* ist.

Beispiel 5.1.12. Die Reduktionsregel

$$\lambda(x.\lambda(y.Z\{x,y\})) \rightarrow \mathtt{0}$$

ist selbst-überlappend, denn für eine Instanziierung mit $s(Z\{x,y\}) = \lambda(z.@(x,y))$ ist $s(\lambda(y.Z\{x,y\})) = \lambda(y.\lambda(z.@(x,y)))$ ein Redex innerhalb eines Termes $s(\lambda(x.\lambda(y.Z\{x,y\}))) = \lambda(x.\lambda(y.\lambda(z.@(x,y))))$, der natürlich auch als Ganzes einen Redex darstellt.

Im Folgenden betrachten wir nur noch orthogonale Termersetzungssysteme höherer Ordnung. Für diese gilt:

Resultat: Orthogonale Termersetzungssysteme höherer Ordnung sind konfluent (vgl. [KOR93, Corollary 13.6]).

5.2 Konstruktoren und Striktheit: Das GDSOS-Format

Definition 5.2.1. Ein Reduktionssystem ist ein Paar $R = (L, \rightarrow)$ mit folgenden Eigenschaften:

- L ist eine Signatur

- $\rightarrow$ ist eine Relation

$\rightarrow$ kann eine beliebige Reduktionsrelation sein. Somit ist also ein Termersetzungssystem höherer Ordnung mit einer Reduktionsrelation $\underset{C}{\rightarrow}$ ein Spezialfall eines Reduktionssystems.

Wir wollen nun als weiteren Spezialfall Reduktionssysteme betrachten, in denen die Reduktion mittels Reduktionsregeln definiert sind, die dem GDSOS-Format aus [San97] genügen. Das GDSOS-Format enthält dabei spezielle Regeln für strikte Positionen, die

auch zur Modellierung von Call-by-Value-Sprachen dienen können. Da Unterterme auf strikten Positionen ausgewertet sein müssen, bevor eine Reduktionsregel auf den Ausdruck selbst angewendet werden darf, sind natürlich auch spezielle Betrachtungen zu (aus Konstruktoren bestehenden) Werten nötig. Wie wir feststellen werden, kann man die GDSOS-Regeln auch mit einem Termersetzungssystem höherer Ordnung darstellen, wenn man die Reduktion, statt in beliebigen Kontexten, ausschließlich in Reduktionskontexten betrachtet und man bei strikten Positionen fordern kann, dass die entsprechenden Metavariablen nur zu Werten instanziiert werden dürfen.

5.2.1 Werte, Metawerte und strikte Positionen

Wir wollen nun genauer beschreiben, wie unsere Werte aussehen sollen, also jene Ausdrücke, die nicht weiter reduziert bzw. ausgewertet werden. Dazu unterteilen wir unsere Operatoren in *kanonische* und *nichtkanonische* Operatoren, bzw. *Konstruktoren* und *Nicht-Konstruktoren*. Werte sollen dabei aus einem Konstruktor-Term (d.h. ein kanonischer Operator mit passenden Operanden) bestehen.

Wie bereits erwähnt, wollen wir Operatoren mit strikten Positionen, die eine Auswertung erfordern, versehen können. Insbesondere möchten wir strikte Positionen bei Bedarf auch für Konstruktoren verwenden, in diesem Fall gilt der Konstruktor-Term erst dann als Wert, wenn alle Operanden an strikten Positionen ausgewertet sind.

Wir benötigen also zunächst einmal eine Erweiterung des Signaturbegriffs, aus der die Konstruktoren und strikten Positionen eindeutig hervorgehen:

Definition 5.2.2. *GDSOS-Signatur*
Als GDSOS-Signatur bezeichnen wir ein Tupel $(O, K, a, strict)$ mit folgenden Eigenschaften:

- (O, a) ist eine Signatur,
- $K \subset O$,
- $\forall F \in O$: $strict(F) \subseteq \{1..n\}$, wobei $a(F)=(k_1,\ldots,k_n)$ und $k_i > 0 \Rightarrow i \notin strict(F)$.

Die Elemente aus $O \setminus K$ sind die Nicht-Konstruktoren und die Elemente aus K die Konstruktoren. Die Menge $strict(F)$, für ein $F \in O$, enthält die strikten Positionen, d.h. ist $i \in strict(F)$, so ist das i-te Argument von F strikt. Ein striktes Argument darf keine Abstraktion sein.

Jetzt können wir eine genaue Definition für Werte angeben:

Definition 5.2.3. Wir definieren die Werte-Menge *Val*, die alle Terme enthält, die wir als Werte zulassen wollen, induktiv wie folgt:

$$\frac{\{t_j \in Val \mid j \in strict(c)}{c(\vec{x}_1.\, t_1,\ldots,\vec{x}_n.\, t_n) \in Val}, \text{ für } c \in K$$

Für den Umgang mit strikten Positionen innerhalb der Regeln brauchen wir noch spezielle Werte-Metavariablen, die nur Werte enthalten dürfen. Werte-Metavariablen haben stets die Stelligkeit 0. Mit Hilfe der Werte-Metavariablen können wir nun Metawerte definieren: Metawerte sind eine bestimmte Menge von Metatermen, die nur für Werte stehen (d.h. alle möglichen Instanziierungen eines Metawertes sind Werte). Für unsere Ableitungsregeln interessieren wir uns besonders für Metaterme, die an allen nicht-strikten Positionen ausschließlich Metavariablen enthalten, da nicht-strikte Positionen nicht ausgewertet werden sollen.

Definition 5.2.4. *Einfache Metawerte*:
Wir definieren die Menge der einfachen Metawerte *Mval* induktiv wie folgt:

$$\frac{}{ValMvar \subseteq Mval}$$

$$\frac{\{M_j \in Mval \mid j \in strict(c)\} \quad \{M_k \in Mvar \mid k \in \{1..n\} \setminus strict(c)\}}{c(M_1,...,M_n) \in Mval}, c \in K$$

5.2.2 Reduktionsregeln

Definition 5.2.5. *Global deterministische, strukturelle, operationale Semantik* (*GDSOS*): Die operationale Semantik einer Sprache L ist im GDSOS-Format, wenn die Reduktionsrelation durch Regeln der folgenden Form ausgedrückt werden kann: Die Regeln für jeden Operator $F \in O$ der Stelligkeit $(k_1,...,k_n)$ mit strikten Positionen $I=strict(F)$ bestehen aus $|I|$ Ableitungsregeln der Form

$$\left\{ \frac{X_i \to Y_i}{F(X_1...X_i...X_n) \to F(X_1...Y_i...X_n)} \right\}_{i \in I},$$

sowie beliebig (möglicherweise unendlich) vielen Axiomen der Form

$$\left\{ \frac{}{F(M_{1j},...,M_{nj}) \to M_j} \right\}_{j \in J},$$

wobei J eine Indizierungsmenge ist und $J = \emptyset$, falls F ein Konstruktor ist.

Des weiteren fordern wir (für alle $i \in I, j \in J$):

(1) M_{ij} ist ein einfacher Metawert, andere Operanden M_{hj}, $h \notin i$, sind Metavariablen.

(2) Die Metavariablen X_i und Y_i sind voneinander verschieden und alle Metavariablen und Werte-Metavariablen erscheinen höchstens einmal in einem Metaterm außer M_j.

(3) Die Menge der linken Seiten der Axiome ist nicht-überlappend (d.h. keine zwei linken Seiten können durch eine Instanziierung unifiziert werden).

(4) Es gibt keine freien (gewöhnlichen) Variablen in M_j.

(5) Alle Metavariablen in M_j sind in einem M_{kj} enthalten ($k \in 1...n$).[33]

Ein *call-by-value* GDSOS - Format liegt vor, wenn alle Meta-Applikationen $X\{N_1...N_n\}$ in M_j der Art gestaltet sind, dass die geschlossenen Instanzen von N_i zwingend Werte sind.

Die $|I|$ Ableitungsregeln erlauben die Auswertung in jedem strikten Argument. Die Nebenbedingungen dienen folgenden Zielen:

(1) Die Axiome sollen erst angewendet werden können, wenn alle strikten Argumente zu einem Wert ausgewertet wurden. Die Definition der einfachen Metawerte stellt sicher, dass der Auswertungsschritt nur von den Konstruktoren dieses Wertes abhängt.

(2) Die Regeln sollen nicht von der syntaktischen Äquivalenz von willkürlichen Termen abhängen.

(3) - (5) Die Regeln sollen deterministisch sein.

Wir können die Axiome als Reduktionsregeln eines Termersetzungssystems höherer Ordnung darstellen, wenn wir Operanden, bestehend aus einer Metavariablen X, mit $a(X) = n$ und $n > 0$, umschreiben in $x_1...x_n \,.X\{x_1,..., x_n\}$; dies entspricht einer Umbenennung der gebundenen Variablen. Wie man sieht, genügen die Axiome nun Definition 5.1.5., bis auf die Tatsache, dass sie evtl. Werte-Metavariablen enthalten.

Auf die Ableitungsregeln für strikte Positionen können wir nun verzichten, wenn man die durch die Axiome beschriebenen unmittelbaren Reduktionsschritte in Reduktionskontexten ausführen darf:

Definition 5.2.6. Reduktionskontexte sind:

$$R ::= [] \;\mid\; F(\vec{x}_1.\, t_1, ..., \vec{x}_{i-1}.\, t_{i-1}, R, \vec{x}_{i+1}.\, t_{i+1}, ..., \vec{x}_n \,.\, t_n), \text{ für alle } F \in O \text{ und strikte Positionen } i \in \text{strict}(F).$$

Da ein Normalordnungsredex entweder der ganze Term ist oder ein Unterterm an einer strikten Position, wird somit in Bezug auf die Reduktion der strikten Argumente dieselbe Wirkung erzielt, wie bei den Ableitungsregeln.

5.2.3 Beweis-Prinzipien für GDSOS-Reduktionssysteme

Definition 5.2.7. Ein *GDSOS-Reduktionssystem* ist ein Paar $(L, \rightarrow)$ mit folgenden Eigenschaften:

- L ist eine GDSOS-Signatur.

[33] Dies wird in [San97] nicht explizit gefordert, ist aber im Sinne einer Determinismus-Forderung nötig.

- $\rightarrow$ ist eine Reduktionsrelation, definiert durch GDSOS-Regeln über L.

Die Auswertung eines GDSOS-Reduktionssystems ist gegeben durch

$$t \downarrow v, \text{ gdw. } t \rightarrow^* v, v \in Val.$$

Sands zeigt in [San97] die Gültigkeit spezieller Beweismethoden:

1. Syntaktische Kontinuität für rekursive Konstanten ist gültig und kann somit als Basis für eine Fixpunkt-Induktion verwendet werden,

2. „Verbesserungs"-Induktion[34] kann für spezielle „instrumentierte" GDSOS-Semantiken verwendet werden,

3. und eine verallgemeinerte, auf Koinduktion basierende Version der applikativen Bisimulation, wie für den „lazy" Lambda-Kalkül beschrieben, ist eine Kongruenz und stimmt unter bestimmten Umständen mit der kontextuellen Gleichheit überein.

Betrachten wir kurz die entsprechenden Resultate, ohne auf die Details einzugehen:

1. Fixpunkt-Eigenschaften:

Sei f eine rekursive Konstante, definiert durch $f = C_f[f]$, für einen geschlossenen Kontext, der f nicht enthält; sei außerdem die Menge der Konstanten $\{f^i\}_{i\geq 0}$ induktiv definiert durch $f^0 = f^0$, $f^{i+1} = C_f[f^i]$: Ein um solche Konstanten erweitertes GDSOS-Reduktionssystem nennen wir GDSOS-Reduktionssystem mit Rekursion. Die rekursiven Konstanten dienen dabei lediglich einer einfacheren Betrachtung, da die Konstanten nicht wie Variablen eingefangen werden können, die Mächtigkeit des Systems wird dadurch nicht erweitert, wenn bereits nichtterminierende Terme existieren (vgl. [San97, Fußnote 2]).

Die Funktionen $\{f^i\}_{i\geq 0}$ bilden in der kontextuellen Ordnung eine aufsteigende Kette, beschränkt durch f, d.h. es gilt: $f =_C C_f[f]$ und $\forall\, i \geq 0: f^i \leq_C f^{i+1} \leq_C f$.

Sands zeigt die folgenden Eigenschaften:

Resultat (Syntaktische Kontinuität): Für alle Kontexte $C[]$ und Terme t:

$$C[f] \leq_C t \Leftrightarrow \forall\, i \geq 0: C[f^i] \leq_C t.$$

Resultat (Kleinster Prä-Fixpunkt): Sei t ein geschlossener Term, dann gilt:

$$C_f[t] \leq_C t \Rightarrow f \leq_C t.$$

[34] engl. *improvement induction*

2. „Verbesserungs“-Induktion:

Diese Beweismethode betrachtet spezielle „instrumentierte“ GDSOS-Reduktionssysteme, deren Reduktionsregeln mit einer Ressourceninformation versehen sind. Eine „Verbesserung“ ist eine Erweiterung der kontextuellen Approximation, so dass s durch t verbessert wird, falls für alle Kontexte gilt: Falls $C[s]$ terminiert, dann terminiert auch $C[t]$ mit geringerem oder gleich großem Ressourcenbedarf. Ein instrumentiertes GDSOS-Reduktionssystem ist *monoton*, wenn die Verknüpfung der Ressourceninformation bei Hintereinanderausführung verschiedener Regeln eine Präordnung $\geq$ erhält, und es ist *wohldefiniert monoton* (oder nur *wohldefiniert*), wenn $>$, definiert durch $\geq \cap \neq$, wohldefiniert ist und die Verknüpfung $>$-monoton ist (für Details siehe [San97]). Die kontextuelle Approximation ist ein Spezialfall der Verbesserung, der entsteht, wenn der Ressourcenverbrauch aller Reduktionen als 0 definiert wird; diese Verbesserung ist monoton, aber nicht wohldefiniert, da der Ressourcenverbrauch bei Hintereinanderausführung verschiedener Regeln nicht steigt.

Seien folgende Relationen definiert:

- $s \geq_I t$, gdw. s durch t verbessert wird.
- $s \sim_I t$ gdw. $s \geq_I t \wedge t \geq_I s$.
- $s \geq_I^r t$ gdw. Terme t_1 und t_2 existieren, so dass: $s \geq_I t_1$, t_1 reduziert mit dem Bedarf von r Ressourcen zu t_2 und $t_2 \geq_I t$.
- $s \sim_I^r t$ gdw. Terme t_1 und t_2 existieren, so dass: $s \sim_I t_1$, t_1 reduziert mit dem Bedarf von r Ressourcen zu t_2 und $t_2 \sim_I t$.

Dann gilt:

Resultat (Verbesserungs-Induktion): Für jedes wohldefinierte instrumentierte GDSOS-Reduktionssystem, für alle geschlossenen C, s, t und für alle Ressourcen $t > 0$ gilt:

$$\left.\begin{array}{l} s \geq_I^r C[s] \\ t \sim_I^r C[t] \end{array}\right\} \Rightarrow s \geq_I t\,.$$

3. Bisimulation:

Sands definiert eine *offene Verbesserungs-Simulation*[35] i für instrumentierte GDSOS-Reduktionssysteme, so dass, falls $s\ i\ t$, für alle abschließenden Substitutionen s gilt:

Falls $s(s) \downarrow c(\vec{x}_1.s_1,...,\vec{x}_n.s_n)$, mit einem Ressourcenbedarf von r_s, dann $s(t) \downarrow c(\vec{x}_1.t_1,...,\vec{x}_n.t_n)$, mit einem Ressourcenbedarf von r_t, so dass $r_s \geq r_t$ und für alle i, mit $1 \leq i \leq n$, gilt: $s_i\ i\ t_i$.

[35] engl. *open improvement simulation*

Resultat: Für jedes monotone instrumentierte GDSOS-Reduktionssystem gilt: Falls i eine offene Verbesserungs-Simulation ist, dann ist $i \subseteq \geq_I$. Im Rahmen des Beweises wird gezeigt, dass die *maximale Verbesserungs-Simulation*, gegeben durch die Vereinigung aller Verbesserungs-Simulationen, eine Präkongruenz ist.

Die maximale Verbesserung ist somit *koinduktiv* als größte Verbesserungs-Simulation definiert. Koinduktion kann man als Gegenstück zur Induktion sehen, denn während die Induktion über den kleinsten Fixpunkt definiert, ist die Koinduktion über den größten Fixpunkt definiert. Für eine Einführung in koinduktive Beweistechniken in Zusammenhang mit funktionaler Programmierung, siehe [Gor94].

Falls wir nun jedem Reduktionsschritt einen Resourcenverbrauch 0 zuordnen und somit ein monotones instrumentiertes GDSOS-Reduktionssystem erhalten, entspricht die maximale Verbesserungs-Simulation einer Verallgemeinerung der Simulations-Relation $\leq_b$ aus Abschnitt 3.2.2 (allerdings seitenvertauscht). Wie bereits bei der Vorstellung der „Verbesserungs"-Induktion beschrieben, entspricht in diesem Fall auch die Relation $\geq_I$ der (seitenvertauschten) kontextuellen Approximation $\leq_C$. Dieser Speziallfall wird in [How96] ausführlicher behandelt: Insbesondere wird dort untersucht, wann eine verallgemeinerte Bisimulation mit der kontextuellen Gleichheit übereinstimmt, also nicht nur enthalten ist. Sands verweist für diese Betrachtung ebenfalls auf [How96].

5.3 Strukturierte Auswertungssysteme

Wir betrachten daher das Regelformat aus [How96] genauer. Die dort definierten *strukturierten Auswertungssysteme* beschreiben die operationale Semantik mittels einer Auswertungsrelation, also als „big step"-Semantik. Wie wir sehen werden, lässt sich ein Reduktionssystem mit GDSOS-Reduktionsregeln in ein (gemäß Auswertung äquivalentes) strukturiertes Auswertungssystem übersetzen. Strukturierte Auswertungssysteme sind jedoch allgemeiner, da sie auch nicht-deterministische Auswertungen zulassen.

Definition 5.3.1. Ein *Auswertungssystem* ist ein Paar $e = (L, \Downarrow)$ mit folgenden Eigenschaften:

- L ist eine Signatur.
- $\Downarrow \subset T_0 \times T_0$, und für alle $t, v \in T_0$, falls $t \Downarrow v$, dann gilt für alle v': $v \Downarrow v'$ gdw. $v \equiv v'$.

Die Relation $\Downarrow$ bezeichnen wir als *Auswertungsrelation* des Auswertungssystems. Ein *Wert* ist ein Term $v \in T_0$ für den $v \Downarrow v$ gilt, d.h. ein Wert wird nicht weiter ausgewertet. Aufgrund der Definition von $\Downarrow$ gilt ebenfalls: v ist ein Wert gdw. $\exists\, t \in T_0$: $t \Downarrow v$. d.h. ein Wert ist das Ergebnis einer Auswertung.

5.3.1 Auswertungsregeln

Definition 5.3.2. Ein *einfacher Metaterm* hat die Form $F(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\})$, wobei die X_i voneinander verschiedene Metavariablen sind.

Definition 5.3.3. *Strukturierte Auswertungsregeln*:

Eine strukturierte Auswertungsregel ist eine Ableitungsregel, deren Formeln alle die Form $M \Downarrow N$ haben, wobei M und N Metaterme ohne freie Variablen sind. Insbesondere besteht eine Auswertungsregel r aus einer Menge I_r, einer vollständigen, wohlfundierten[36] Relation $\leq_r$ über I_r, einer Formel-Familie $\{M_i \Downarrow N_i\}$, $i \in I_r$ indiziert durch I_r und einer Formel $M \Downarrow N$, die wir *Schluss* von r nennen. Die Formeln $M_i \Downarrow N_i$, $i \in I_r$ nennen wir *Prämissen* von r. Zusätzlich benötigen wir das Folgende:

(1) M ist ein einfacher Metaterm.

(2) Für alle $i \in I_r$ ist N_i eine Metavariable oder ein einfacher Metaterm und hat keine Metavariablen mit M oder einem N_j für ein $j \neq i$ gemeinsam.

(3) Für alle $i \in I_r$ und jede Metavariable X, die in M_i vorkommt, kommt X bereits in M vor oder in einem N_j für ein $j \leq_r i$. Jede Metavariable in N kommt in einem N_j vor, für ein $i \in I_r$.

(4) Für jede Metavariable X, die in der Auswertungsregel vorkommt, ist X eine Werte-Metavariable gdw. X ein N_i für ein $i \in I_r$ ist.

(5) Wenn N eine Metavariable ist, ist sie ein N_i für ein $N_i \in I_r$.

Eine solche Regel nennen wir Regel für F, falls F der Operator in M ist.

Eine Instanz einer Auswertungsregel ist das Ergebnis der Anwendung einer Instanziierung σ, deren Gültigkeitsbereich die Metavariablen in der Regel enthält, auf die Prämissen und den Schluss der Regel.

Sei R eine Menge von Auswertungsregeln. *Herleitungen* über R sind Bäume von Instanzen von Regeln aus R, wobei die Kinder eines Knotens r in einer Eins-zu-eins-Beziehung mit den Prämissen von r stehen und der Schluss jeden Kindes derselbe ist, wie in der korrespondierenden Prämisse. Die Instanz der Regel an der Wurzel nennen wir den *letzten Schritt* der Herleitung. Der *Schluss* einer Herleitung ist der Schluss des letzten Schrittes. Eine Herleitung $\tilde{N}$ ist eine Herleitung *für* einen Term t, falls $t \Downarrow v$ der Schluss von $\tilde{N}$ für ein v ist.

Definition 5.3.4. *Strukturiertes Auswertungssystem*:

Ein strukturiertes Auswertungssystem besteht aus einem Auswertungssystem e und einer Menge R von strukturierten Auswertungsregeln, deren Formeln aus Metatermen über e bestehen, so dass die Auswertungsrelation von e gleich ist mit der Relation $\Downarrow$, definiert durch $s \Downarrow t$, falls die Formel $s \Downarrow t$ eine Herleitung über R hat. Ein strukturiertes Auswertungssystem ist endlich, falls es eine endliche Anzahl von Regeln für jeden Operator gibt und jede Regel eine endliche Anzahl von Prämissen hat.

[36] d.h. es gibt keine Teilmenge von I_r, die bzgl. der Relation $\leq_r$ eine unendlich absteigende Kette bildet.

Damit eine Menge von Auswertungsregeln ein strukturiertes Auswertungssystem beschreibt, muss gemäß Definition 5.3.1. gelten: $\forall\ t, v \in T_0$, falls $t \Downarrow v$, dann gilt für alle v': $v \Downarrow v'$ gdw. $v \equiv v'$. Sind spezielle Regeln für kanonische Operatoren enthalten, so folgt dies unmittelbar:

Lemma 5.3.5. Eine Menge R von Auswertungsregeln beschreibt ein strukturiertes Auswertungssystem, falls es eine Untermenge von Operatoren $K \subset O$ gibt, die die folgenden beiden Bedingungen erfüllt:

- Falls $M \Downarrow N$ der Schluss einer Regel für ein $F \in O \setminus K$ ist, dann ist N eine Metavariable.

- Für jedes $c \in K$ gibt es genau eine Regel für c in R, diese hat die Form

$$\frac{\{X_i \Downarrow Y_i\}_{1 \le i \le n, X_i \neq Y_i}}{c(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\}) \Downarrow c(\vec{x}_1.Y_1\{\vec{x}_1\},...,\vec{x}_n.Y_n\{\vec{x}_n\})}$$, wobei für jedes i, $1 \le i \le n$, $X_i = Y_i$ ist oder die Stelligkeit von X_i ist 0.

(vgl. [How96, Lemma 6.1])

5.3.2 Eigenschaften strukturierter Auswertungssysteme

Betrachten wir nun die in [How96] definierte Verallgemeinerung der Bisimulation. Wie bereits beschrieben, interessieren wir uns insbesondere dafür, wann die Bisimulation mit der kontextuellen Gleichheit übereinstimmt.

Definition 5.3.6. Für eine Relation $R \subset T \times T$ definieren wir $[R] \subset T_0 \times T_0$ als $t\ [R]\ t'$, gdw. für alle kanonischen Terme der Form $F(s_1, ..., s_n)$ gilt: Falls $t \Downarrow F(s_1, ..., s_n)$, dann existieren Terme $s_1', ..., s_n'$, so dass $t' \Downarrow F(s_1', ..., s_n')$ und für alle i, mit $1 \le i \le n$: $s_i\ R\ s_i'$.

Definition 5.3.7. Für eine Relation $R \subset T_0 \times T_0$ definieren wir die *Erweiterung auf offene Terme* R° als $t\ R^\circ\ t'$, gdw. $s(t)\ R\ s(t')$, für jede, die Terme abschließende Substitution s. Außerdem definieren wir für $R \subset T \times T$ die *Einschränkung auf geschlossene Terme* R_0 als $R_0 = R \cap (T_0 \times T_0)$.

Definition 5.3.8. Sei $R \subset T_0 \times T_0$: R ist eine *Simulation*, gdw. R eine Präordnung ist und $R \subset [R^\circ]$.

Wir sehen also, dass dieser Begriff der Simulation der offenen Verbesserungs-Simulation bei einem Ressourcenverbrauch von 0 für alle Regeln entspricht.

Definition 5.3.9. Wir definieren die Relation $\le_B$ koinduktiv als größte Simulation und definieren die Relation $\sim_B$ als $s \sim_B t$ gdw. $s \le_B t \wedge t \le_B s$.[37]

[37] In [How96] wird $\sim_B$ als symmetrische Hülle von $\le_B$ definiert. Da sich aber alle für uns relevanten Beweise auf $\le_B$ beziehen, können wir hier $\sim_B$ etwas aussagekräftiger definieren.

Beispiel 5.3.10. Für eine Signatur L_Λ, wie in Beispiel 5.1.3. beschrieben, entspricht $\sim_B$ der applikativen Bisimulation des „lazy“ Lambda-Kalküls.

Für strukturierte Auswertungssysteme gilt folgende Eigenschaft:

Satz 5.3.11. Falls ε ein strukturiertes Auswertungssystem ist, dann ist $\leq_B$ eine Präkongruenz.

Beweis: Siehe [How96, Collary 6.1].

Howe definiert noch eine weitere Verallgemeinerung der Bisimulation zur Behandlung von Divergenz bei nicht-deterministischen Auswertungen. Für deterministische Auswertungen reicht die Betrachtung von $\sim_B$.

Betrachten wir nun die Beziehung zwischen $=_C$ und $\sim$:

Definition 5.3.12. Sei ε ein strukturiertes Auswertungssystem: ε ist *deterministisch*[38], gdw. $s \equiv t$, falls $r \Downarrow s$ und $r \Downarrow t$. ε ist *berechungsvollständig*[39], gdw. es für alle F, mit $a(F) = (k_1, ..., k_n)$, und für alle i, $1 \leq i \leq n$ und alle abschließenden Substitutionen s stets einen Kontext $C_{F,i,s}[]$ gibt, so dass für alle Werte $F(r_1, ..., r_n)$ gilt: Falls r_i die Form $x_1,...,x_{k_i}.t$ hat, dann gilt $C_{F,i,s}[F(r_1, ..., r_n)] \sim s(t)$.

Satz 5.3.13. (Kontext-Lemma) Falls ε ein deterministisches, berechnungsvollständiges Auswertungssystem ist, $\sim$ eine Kongruenz ist und $\leq_C \subset (\leq_C)_0^\circ$, dann gilt:

$$=_C = \sim^\circ.$$

Beweis: Siehe [How96, Theorem 5.1].

5.3.3 Die Übersetzung der GDSOS-Regeln in strukturierte Auswertungsregeln

Wie man sieht, stellt ein GDSOS-Reduktionssystem mit der Auswertung $\downarrow$ ein Auswertungssystem dar. Im Folgenden soll nun gezeigt werden, dass ein GDSOS-Reduktionssystem in ein äquivalentes strukturiertes Auswertungssystem übersetzt werden kann, so dass $s \downarrow t$ gdw. $s \Downarrow t$. Dazu definieren wir eine Übersetzungsfunktion f, mit der wir ein GDSOS-Reduktionssystem $S = (L, \rightarrow)$ in ein Auswertungssystem $f(S) = (\varepsilon, R)$ übersetzen können.

Wir überzeugen uns zunächst davon, dass das übersetzte System ein strukturiertes Auswertungssystem ist, insbesondere erfüllen die Auswertungsregeln R die Bedingungen aus Definition 5.3.3. Anschließend beweisen wir die Äquivalenz der Auswertung, indem wir zunächst zeigen, dass aus $s \downarrow t$ auch $s \Downarrow t$ folgt, und anschließend die Rückrichtung $s \Downarrow t \Rightarrow s \downarrow t$ beweisen.

---

[38] Howe benutzt hier das Wort *determinate*, was mit *bestimmt*, *begrenzt*, *festgelegt* oder *entschieden* jedoch nicht besonders treffend übersetzt werden kann.
[39] engl. *computationally complete*

Definition 5.3.14. Die Übersetzungsfunktion f:
Sei $S = (L, \rightarrow)$ ein GDSOS-Reduktionssystem und $L = (O, K, a, strict)$, und sei R' eine Abänderung der GDSOS-Regeln für $\rightarrow$, in der gerade jede Metavariable X, mit $a(X) = n$ und $n > 0$, durch einen Metaterm der Form $x_1, ..., x_n.X\{x_1, ..., x_n\}$ ersetzt wurde:

Die Übersetzungsfunktion f ist definiert durch $f(S) = (\varepsilon, R)$, mit $\varepsilon = (L, \Downarrow)$, so dass $s \Downarrow t$ gdw. die Formel $s \Downarrow t$ eine Herleitung über R hat. Die Menge der Regeln R wird wie folgt konstruiert:

Für jeden Konstruktor $c \in K$ mit $a(c) = (k_1,...,k_n)$, $strict(c) = I$ enthält R genau eine Regel

$$\frac{\{X_i \Downarrow Y_i\}_{i \in I}}{c(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\}) \Downarrow c(\vec{x}_1.Y_1\{\vec{x}_1\},...,\vec{x}_n.Y_n\{\vec{x}_n\})}, \text{ wobei } X_k = Y_k, \text{ für alle } k \neq i.$$

Für alle Operatoren $F \in O \setminus K$ mit $a(F) = (k_1,...,k_n)$, $strict(F) = I$, enthält R eine Regel

$$r_j = \frac{\{N_{ij} \Downarrow M'_{ij}\}_{i \in I} \quad \{P_{vj}\}_{v \in I_v} \quad M_j \Downarrow V_j}{F(N_{1j},...,N_{nj}) \Downarrow V_j},$$

gdw. R' eine Regel

$$r_j' = \frac{}{F(M_{1j},..., M_{nj}) \rightarrow M_j}$$

enthält, wobei V_j eine neue Werte-Metavariable ist, alle N_{ij} mit $i \in I$ neue (Nicht-Werte-) Metavariablen der Stelligkeit 0 sind und $M_{kj} = N_{kj}$ für alle $k \neq i$. Die Prämissen $\{N_{ij} \Downarrow M'_{ij}\}_{i \in I}$ und $\{P_{vj}\}_{v \in I_v}$ konstruieren wir wie folgt:

Falls M_{ij} ein einfacher Metawert in Form einer Werte-Metavariable ist, so ist $M'_{ij} = M_{ij}$. Ansonsten ist M_{ij} ein einfacher Metawert der Form $c(m_1,...,m_a)$, wobei c ein Konstruktor ist, mit $|\, a(c) \,| = a$, $strict(c) = I_c$ und alle m_k mit $k \in \{1...a\} \setminus I_c$ sind Metavariablen (in der Schreibweise $\vec{x}.X\{\vec{x}\}$), während alle m_l mit $l \in I_c$ einfache Metawerte sind: In diesem Fall definieren wir M'_{ij} als $c(n_1,...,n_a)$, wobei $m_k = n_k$ für alle $k \in \{1 \ldots n_c\} \setminus I_c$, und alle n_l mit $l \in I_c$ sind neue (Nicht-Werte-) Metavariablen. Zusätzlich führen wir neue Prämissen $n_l \Downarrow m'_l$ für alle $l \in I_c$ ein, wobei die m'_l aus der rekursiven Anwendung des Verfahrens auf die einfachen Metawerte m_l hervorgehen.

Da einfache Metawerte nur endlich tief verschachtelt sein können, terminiert das Verfahren und wir erhalten somit die gewünschten Prämissen $\{N_{ij} \Downarrow M'_{ij}\}_{i \in I}$ und $\{P_{vj}\}_{v \in I_v}$. Mit P_{vj} sollen dabei die nach obigem Verfahren hinzugefügten Prämissen der Form $n_l \Downarrow m'_l$ bezeichnet werden und die Indizierungsmenge I_v ergibt sich als $I_v = \{1..a_p\}$, wobei a_p die Anzahl der auf diese Weise erstellten Prämissen ist.

Anmerkung:

Die Ableitungsregeln der Form $\left\{\frac{X_i \to Y_i}{F(X_1...X_i...X_n) \to F(Y_1...Y_i...Y_n)}\right\}_{i\in I}$ für nicht-kanonische Operatoren fallen weg, da die Prämissen $\{N_{ij} \Downarrow M'_{ij}\}_{i\in I}$ der übersetzten Regel nun für die Auswertung der strikten Position sorgen.

Lemma 5.3.15. Sei $S = (L, \to)$ ein GDSOS-Reduktionssystem und $f(S) = (\varepsilon, R)$: Dann ist R eine Menge strukturierter Auswertungsregeln.

Beweis: Es ist leicht zu sehen, dass alle Regeln für kanonische Operatoren strukturierte Auswertungsregeln sind, da sie gemäß Lemma 5.3.5. konstruiert wurden. Sei also r_j eine Regel der Form $r_j = \frac{\{N_{ij} \Downarrow M'_{ij}\}_{i\in I} \quad \{P_{vj}\}_{v\in I_v} \quad M_j \Downarrow V_j}{F(N_{1j},...,N_{nj}) \Downarrow V_j}$ für einen nichtkanonischen Operator. Alle Prämissen und der Schluss in r haben die Form $M \Downarrow N$, wobei M und N Metaterme ohne freie Variablen sind, da die M_{ij} bzw. M_j gemäß der GDSOS-Bedingung keine freien Variablen haben und durch die Übersetzung keine neuen freien Variablen eingeführt werden. Wir überprüfen nun die Bedingungen 1-5 des Formats der strukturierten Auswertungsregeln für diese Regel:

Bedingung 1) z.z.: $F(N_{1j},...,N_{nj})$ ist ein einfacher Metaterm.

Gemäß Übersetzung sind alle N_{kj}, $1 \le k \le n$, Metavariablen in der Schreibweise $\vec{x}_{kj}.X_{kj}\{\vec{x}_{kj}\}$. Somit ist die Bedingung erfüllt.

Bedingung 2) z.z: Die rechten Seiten der Prämissen sind Metavariablen oder einfache Metaterme und haben keine gemeinsamen Metavariablen.

Aufgrund der GDSOS-Bedingungen gilt: Alle Metavariablen und Werte-Metavariablen kommen höchstens einmal in einem der M_{ij} vor. Die von der Übersetzung neu eingeführten Metavariablen kommen ebenfalls höchstens in einer Prämisse auf der rechten Seite vor. Zudem haben wir die M'_{ij} und die rechten Seiten der $\{P_{vj}\}_{v\in I_v}$ so konstruiert, dass diese jeweils entweder eine Werte-Metavariable sind, oder ein Metaterm der Form $c(n_1,...,n_a)$, wobei alle Operanden an strikten Positionen neu eingeführte Metavariablen sind und alle anderen Operanden ebenfalls Metavariablen sind, da diese aus den nicht-strikten Positionen eines einfachen Metawertes übernommen wurden.

Bedingung 3) z.z.: Jede Metavariable auf der linken Seite einer Prämisse ist ein N_{kj}, für $1 \le k \le n$, oder kommt auf der rechten Seite einer vorhergehenden Prämisse vor.

Wir haben die Übersetzung so konstruiert, dass dies für alle Metavariablen auf der linken Seite einer Prämisse automatisch erfüllt ist, mit Ausnahme der Metavariablen innerhalb M_j. Für diese gilt jedoch aufgrund der GDSOS-Bedingungen: Alle Metavariablen in M_j sind in einem M_{kj} enthalten.

Bedingung 4) z.z.: Eine Metavariable ist genau dann eine Werte-Metavariable, wenn es eine Prämisse gibt, für die sie der Metaterm auf der rechten Seite ist.

Jede Werte-Metavariable kommt in der ursprünglichen Regel an einer strikten Position vor. Für diese haben wir jeweils eine neue Prämisse erstellt, so dass die Werte-Metavariable der Metaterm auf der rechten Seite ist.

Bedingung 5) z.z. Es gibt eine Prämisse, für die V_j der Metaterm auf der rechten Seite ist.

Die Regel enthält gemäß Konstruktion die Prämisse $M_j \Downarrow V_j$.

Äquivalenz der Auswertung

z.z.: $s \downarrow t \Leftrightarrow s \Downarrow t$

$\Rightarrow$: Wir zeigen die Behauptung durch Induktion über die Anzahl der Reduktionsschritte. Dazu benötigen wir zunächst einige Aussagen über Werte. Wir nehmen dabei einen Teil der Rückrichtung vorweg und beweisen die Äquivalenz bei Werten. Anschließend können wir uns dem Induktionsschritt widmen.

Lemma 5.3.16. Sei S ein GDSOS-Reduktionssystem und $f(S) = (\varepsilon, R)$:
$(\exists\, t \in T_0 \colon t \Downarrow v) \Rightarrow v \in Val$.

Beweis: Durch Induktion über die Struktur von $\Downarrow$.

Induktionsanfang: Regeln der Form $\dfrac{}{M \Downarrow N}$.

Jede Regel für einen nichtkanonischen Operator hat mindestens eine Prämisse (nämlich $M_j \Downarrow V_j$). Die Regeln für einen kanonischen Operator haben für jede strikte Position eine Prämisse. Somit kommt nur eine Regel

$$\frac{}{c(\vec{x}_1.X_1\{\vec{x}_1\},\ldots,\vec{x}_n.X_n\{\vec{x}_n\}) \Downarrow c(\vec{x}_1.X_1\{\vec{x}_1\},\ldots,\vec{x}_n.X_n\{\vec{x}_n\})},$$

für ein $c \in K$, mit $a(c) = (k_1,\ldots,k_n)$ und $strict(c) = \{\}$, in Frage. Jede Instanz $s\,(c(\vec{x}_1.X_1\{\vec{x}_1\},\ldots,\vec{x}_n.X_n\{\vec{x}_n\})) = c(\vec{x}_1.t_1,\ldots,\vec{x}_n.t_n)$ ist ein Element von Val.

Induktionsvoraussetzung: Für Regeln der Form $\dfrac{\{M_i \Downarrow V_i\}}{M \Downarrow V} \in R$ gilt $s(V_i) \in Val$, für alle Instanziierungen s.

Induktionsschritt: z.z.: Für alle Regeln der Form $r = \dfrac{\{M_i \Downarrow V_i\}}{M \Downarrow V} \in R$ gilt $s(V) \in Val$, für alle Instanziierungen s.

Fall 1: r ist eine Regel für einen nichtkanonischen Operator F. Dann hat r die Form

$$\frac{\{N_{ij} \Downarrow M'_{ij}\}_{i\in I} \quad \{P_{vj}\}_{v\in I_V} \quad M_j \Downarrow V_j}{F(N_{1j},\ldots,N_{nj}) \Downarrow V_j}.$$

Aufgrund der Induktionsvoraussetzung ist $s(V_j) \in Val$.

Fall 2: r ist eine Regel für einen kanonischen Operator c. Dann hat r die Form

$$\frac{\{X_i \Downarrow Y_i\}_{i\in I}}{c(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\}) \Downarrow c(\vec{x}_1.Y_1\{\vec{x}_1\},...,\vec{x}_n.Y_n\{\vec{x}_n\})}, \text{ wobei } X_k = Y_k, \text{ für alle } k \neq i.$$

Da alle $s(Y_i)$ an strikten Positionen gemäß Induktionsvoraussetzung Elemente von Val sind, ist gemäß Definition von Val auch

$$s\,(c(\vec{x}_1.Y_1\{\vec{x}_1\},...,\vec{x}_n.Y_n\{\vec{x}_n\})) = c(s\,(\vec{x}_1.Y_1\{\vec{x}_1\}),...,s\,(\vec{x}_n.Y_n\{\vec{x}_n\}))$$

ein Element von Val für jede Instanziierung s, denn für alle $i \in I$ ist die Stelligkeit von Y_i gleich 0, d.h. $\vec{x}_i.Y_i\{\vec{x}_i\} = Y_i$.

Lemma 5.3.17. Sei S ein GDSOS-Reduktionssystem und $f(S) = (\varepsilon, R)$: $v \in Val \Leftrightarrow v \Downarrow v$.

Beweis: Wir zeigen $\Rightarrow$ durch Induktion über die Struktur von Val. Die Rückrichtung folgt direkt aus Lemma 5.3.16.

Induktionsanfang: Konstruktor ohne strikte Argumente.

Sei $c \in K$ mit $a(c) = (k_1,...,k_n)$ und $strict(c) = \{\}$. Dann enthält R' eine Regel

$$\frac{}{c(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\}) \Downarrow c(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\})}.$$

Induktionsvoraussetzung: Für alle Terme $c(t_1,...,t_n)$ mit $c \in K$, $a(c) = (k_1,...,k_n)$ und $strict(c) = I$ gilt: Für alle $i \in I$: $t_i \in Val \Leftrightarrow t_i \Downarrow t_i$.

Induktionsschritt:

Gemäß Definition von f enthält R genau eine Regel für c der Form

$$\frac{\{X_i \Downarrow Y_i\}_{i\in I}}{c(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\}) \Downarrow c(\vec{x}_1.Y_1\{\vec{x}_1\},...,\vec{x}_n.Y_n\{\vec{x}_n\})}, \text{ wobei } X_k = Y_k, \text{ für alle } k \neq i.$$

Mit der Induktionsvoraussetzung folgt somit für alle Terme der Form $c(t_1,...,t_n)$:

$c(t_1,...,t_n) \in Val \Leftrightarrow c(t_1,...,t_n) \Downarrow c(t_1,...,t_n)$.

Korollar 5.3.18. Sei S ein GDSOS-Reduktionssystem und $f(S) = (\varepsilon, R)$:
$(\exists\, t \in T_0: t \Downarrow v) \Leftrightarrow v \Downarrow v \Leftrightarrow v \in Val$.

Somit haben wir die Äquivalenz der Werte bewiesen. Für unseren Induktionsschritt des Beweises für $s \downarrow t \Rightarrow s \Downarrow t$ benötigen wir folgende Aussage über strikte Positionen:

Lemma 5.3.19. Sei S ein GDSOS-Reduktionssystem und $f(S) = (\varepsilon, R)$: Falls $t \Downarrow v$ und $t' \Downarrow v$ dann gilt: $F(...t...) \Downarrow c$ gdw. $F(...t'...) \Downarrow c$.

Beweis: Sei $\frac{}{F(M_{1j},...,M_{ij},...,M_{nj}) \rightarrow M_j}$ die passende Regel für F im GDSOS-Format. Dann enthält R die Regel $\frac{\{N_{ij} \Downarrow M'_{ij}\}_{i \in I} \quad \{P_{vj}\}_{v \in I_V} \quad M_j \Downarrow V_j}{F(N_{1j},...,N_{ij},...,N_{nj}) \Downarrow V_j}$.

Angenommen, es gibt Instanziierungen s und s', so dass $s\ (N_{ij}) = t \Downarrow s\ (M'_{ij}) = v$ und $s'\ (N_{ij}) = t' \Downarrow s'\ (M'_{ij}) = v$ und für alle $k \neq i$ gilt $s\ (N_{kj}) = s'\ (N_{kj})$: Da nur eine Prämisse N_{ij} enthält und diese gemäß Voraussetzung erfüllt ist, sind die Prämissen somit entweder in beiden Fällen erfüllt oder gar nicht.

Außerdem benötigen wir eine Aussage über die Prämissen der Form $\{P_{vj}\}_{v \in I_V}$, die bei der Übersetzung der ursprünglichen Regeln für die einfachen Metawerte erstellten wurden:

Lemma 5.3.20. Sei $r_j = \frac{\{N_{ij} \Downarrow M'_{ij}\}_{i \in I} \quad \{P_{vj}\}_{v \in I_V} \quad M_j \Downarrow V_j}{F(N_{1j},...,N_{nj}) \Downarrow V_j}$ die Übersetzung der Regel $r_j' = \frac{}{F(M_{1j},...,M_{nj}) \rightarrow M_j}$, und sei s' eine Instanziierung, die die Anwendung der Regel r_j' erlaubt, d.h. $t_1 = s'(F(M_{1j},...,M_{nj}))$, $t_2 = s'(M_j)$ und $s'(M_{ij}) \in Val$ für alle $i \in I$, $I = strict(F)$. Sei außerdem s eine Instanzierung mit folgenden Eigenschaften:

Für alle $k \in \{1...n\}$ gilt $s(N_{kj}) = s'(M_{kj})$ und für alle $i \in I$ gilt $s(N_{ij}) = s(M'_{ij})$.

Für alle Prämissen $N \Downarrow N' \in \{P_v\}_{v \in I_V}$ gilt $s(N) = s(N')$.

Dann gilt für alle Prämissen $N \Downarrow N' \in \{P_{vj}\}_{v \in I_V}$: $s(N) \Downarrow s(N')$, d.h. $s'(N') \in Val$.

Beweis: Wir zeigen durch Induktion über die Struktur der einfachen Metawerte bzw. über die Rekursionsstufen des Konstruktionsverfahrens der Prämissen:

Für alle Prämissen $N \Downarrow M$, die bei der Übersetzung für M_{ij} erstellt wurden, gilt $s(N) \in Val$.

Induktionsanfang: Der einfache Metawert M_{ij} ist eine Werte-Metavariable oder ein einfacher Metaterm (d.h. der Konstruktor hat keine strikten Positionen). Die für M_{ij} erstellte Prämisse ist $N_{ij} \Downarrow M'_{ij}$, wobei $M'_{ij} = M_{ij}$ gemäß Konstruktion. Somit ist auch M'_{ij} ein einfacher Metawert und daher $s(M'_{ij}) \in Val$.

Induktionsvoraussetzung: Falls M_{ij} von der Form $c(m_1,...,m_a)$ ist, wobei c ein Konstruktor mit $|\ a(c)\ | = a$, $strict(c) = I_c$ ist, und falls M'_{ij} von der Form $c(n_1,...,n_a)$ ist und bei der

Übersetzung für die Operanden m_l mit $l \in I_c$ die Prämissen $\mathcal{P}$ erstellt wurden, d.h. $\{n_l \Downarrow m'_l\}_{l \in I_c} \subseteq \mathcal{P}$, dann gilt: $s(M) \in Val$ für alle $N \Downarrow M \in \mathcal{P}$.

Induktionsschritt:

z.z.: Falls M_{ij} von der Form $c(m_1,...,m_a)$ ist, wobei c ein Konstruktor mit $| a(c) | = a$, $strict(c) = I_c$ ist, und falls M'_{ij} von der Form $c(n_1,...,n_a)$ ist und bei der Übersetzung für M_{ij} die Prämissen $\mathcal{P}$ erstellt wurden, mit $(\{N_{ij} \Downarrow M'_{ij}\} \cup \{n_l \Downarrow m'_l\}_{l \in I_c}) \subseteq \mathcal{P}$, dann gilt $s(M) \in Val$ für alle $N \Downarrow M \in \mathcal{P}$.

Beweis:

Gemäß Induktionsvoraussetzung gilt $s(M) \in Val$ für alle $N \Downarrow M \in \mathcal{P} \setminus (\{N_{ij} \Downarrow M'_{ij}\}$ und gemäß Definition der Instanziierung gilt $s(N) = s(M)$. Für alle $\{n_l \Downarrow m'_l\}_{l \in I_c}$ folgt also $s(n_l) = s(m'_l) \in Val$. Somit ist auch $s(M'_{ij}) = s(c(n_1,...,n_a)) = c(s(n_1),...,s(n_a)) \in Val$.

Wir können nun folgende Aussage zeigen, die den Induktionsschritt unseres Beweises für $s \downarrow t \Rightarrow s \Downarrow t$ darstellt:

Lemma 5.3.21. Sei S ein GDSOS-Reduktionssystem und $f(S) = (\varepsilon, R)$: Falls $t_1 \rightarrow t_2$ und $t_2 \Downarrow v$ dann gilt: $t_1 \Downarrow v$.

Beweis: Durch Induktion über $\rightarrow$.

Induktionsanfang:

Anwendung einer Regel $\dfrac{}{F(M_1,...,M_n) \rightarrow M}$, d.h. $t_1 = s(F(M_1,...,M_n))$, $t_2 = s(M)$ und $s(M_i) \in Val$ für alle $i \in I = strict(F)$.

Die übersetzte Regel hat die Form $\dfrac{\{N_i \Downarrow M'_i\}_{i \in I} \quad \{P_v\}_{v \in I_V} \quad M \Downarrow V}{F(N_1,...,N_n) \Downarrow V}$.

Somit gibt es eine Instanziierung s', so dass $s'(N_j) = s(M_j)$ für alle $j \in \{1...n\}$ und $s'(M) = s(M) = t_2$. Da $s'(N_i) = s(M_i) \in Val$ für alle $i \in I$ gemäß Voraussetzung, folgt $s'(N_i) \Downarrow s'(N_i)$ mit Lemma 5.3.17., außerdem gilt $s'(M) = t_2 \Downarrow v$.

Daher setzen wir folgende Eigenschaften der Instanziierung voraus:

$s'(N_i) := s'(M'_i)$ für alle $i \in I$ und $s'(V) := v$.

Da zudem bei den Prämissen $\{P_v\}_{v \in I_V}$ die Metavariable auf der rechten Seite stets auf der linken Seite einer vorgehenden Regel auf einer strikten Position eines Konstruktorterms

vorkommt, und die erste Prämisse einer solchen Kette $N_i \Downarrow M'_i$ für ein $i \in I$ ist und $s'(M'_i) := s'(N_i) = s(M_i) \in Val$, setzen wir zusätzlich die folgende Eigenschaft voraus:

Für alle $n \Downarrow m \in \{P_v\}_{v \in I_V}$ gilt: $s'(n) := s'(m)$

z.z.: Die Prämissen $\{P_v\}_{v \in I_V}$ sind erfüllt.

Dies folgt aus Lemma 5.3.20.

Die Regel kann also angewendet werden und es folgt $t_1 \Downarrow v$.

Induktionsvoraussetzung: Für Ableitungsregeln der Form

$$\frac{X_i \to Y_i}{F(X_1 ... X_i ... X_n) \to F(X_1 ... Y_i ... X_n)}$$

gilt, wenn $s(Y_i) \Downarrow v_i$, dann auch $s(X_i) \Downarrow v_i$.

Induktionsschritt:

z.z. $F(...x...) \Downarrow v$ wenn $F(...x'...) \Downarrow v$, wobei $x \Downarrow v'$ und $x' \Downarrow v'$.

Dies folgt direkt mit Lemma 5.3.19.

Somit können wir den Induktions-Beweis formal durchführen:

Lemma 5.3.22. Sei S ein GDSOS-Reduktionssystem und $f(S) = (\varepsilon, R)$: Falls $s \downarrow t$, dann hat $s \Downarrow t$ eine Herleitung über R'.

Beweis: Wir zeigen, dass für jede Folge mit n Auswertungsschritten der Form $t_1 \to t_2 \to ... \to t_n \to v$ auch $t_1 \Downarrow v$ gilt, durch Induktion über n:

Induktionsanfang: $n = 0$.

$v \downarrow v \Rightarrow v \Downarrow v$: Folgt direkt aus Lemma 5.3.17.

Induktionsvoraussetzung: Aus $t_1 \to ... \to t_{n-1} \to v$ folgt $t_1 \Downarrow v$.

Induktionsschritt:

z.z.: Aus $t_1 \to t_2$ und $t_2 \Downarrow v$ folgt $t_1 \Downarrow v$.

Folgt direkt aus Lemma 5.3.21.

$\Leftarrow$: Befassen wir uns nun mit der Rückrichtung der gewünschten Aussage $s \downarrow t \Leftrightarrow s \Downarrow t$.

Wir benötigen zunächst eine Art Rückrichtung zu Lemma 5.3.20.:

Lemma 5.3.23. Sei $r_j = \dfrac{\{N_{ij} \Downarrow M'_{ij}\}_{i\in I} \;\; \{P_{vj}\}_{v\in I_v} \;\; M_j \Downarrow V_j}{F(N_{1j},...,N_{nj}) \Downarrow V_j}$ die Übersetzung der Regel $r_j' = \dfrac{}{F(M_{1j},..., M_{nj}) \to M_j}$ und sei s eine Instanziierung, so dass die Regel r_j angewendet werden kann, d.h. für alle Prämissen $M \Downarrow N$ gilt $s(M) \Downarrow s(N)$:

Dann gibt es eine Instanziierung s ', die die Anwendung der Regel r' erlaubt, und s '$(M_j) = s(M_j)$ sowie s '$(M_{ij}) = s(M'_{ij})$ für alle $i \in I$ und s '$(M_{kj}) = s(N_{kj})$ für alle $k \in \{1..n\} \setminus I$.

Beweis: Wir definieren s ' so, dass s '$(M_{kj}) = s(N_{kj})$ für alle $k \in \{1..n\} \setminus I$ und s '$(X) = s(X)$ für alle übrigen Metavariablen X. Da die M_{ij} für alle $i \in I$ nur Metavariablen enthalten, die auch in M'_{ij} oder den Prämissen $\{P_{vj}\}_{v\in I_v}$ enthalten sind, kann somit r_j' instanziiert werden.

z.z.: s '$(M_{ij}) = s(M'_{ij})$ für alle $i \in I$.

Wir zeigen durch Induktion über die Struktur der einfachen Metawerte bzw. über die Rekursionsstufen des Konstruktionsverfahrens der Prämissen:

Falls für alle Prämissen $M \Downarrow N$, die bei der Übersetzung für M_{ij} erstellt wurden, $s(M) \Downarrow s(N)$ gilt, dann gilt $s(M_{ij}) = s(M'_{ij})$.

Induktionsanfang: Der einfache Metawert M_{ij} ist eine Werte-Metavariable oder ein einfacher Metaterm (d.h. der Konstruktor hat keine strikten Positionen). Die für M_{ij} erstellte Prämisse ist $N_{ij} \Downarrow M'_{ij}$, und gemäß Konstruktion gilt $M'_{ij} = M_{ij}$.

Induktionsvoraussetzung: Falls M_{ij} von der Form $c(m_1,...,m_a)$ ist, wobei c ein Konstruktor mit $|\, a(c) \,| = a$, $strict(c) = I_c$ ist, und falls M'_{ij} von der Form $c(n_1,...,n_a)$ ist und bei der Übersetzung für die Operanden m_l mit $l \in I_c$ die Prämissen $\mathcal{P}$ erstellt wurden, d.h. $\{n_l \Downarrow m'_l\}_{l\in I_c} \subseteq \mathcal{P}$, und für alle $M \Downarrow N \in \mathcal{P}$ gilt $s(M) \Downarrow s(N)$, dann gilt $s(m'_l) = s(m_l)$ für alle $l \in I_c$.

Induktionsschritt:

z.z.: Falls M_{ij} von der Form $c(m_1,...,m_a)$ ist, wobei c ein Konstruktor mit $|\, a(c) \,| = a$, $strict(c) = I_c$ ist, und falls M'_{ij} von der Form $c(n_1,...,n_a)$ ist und bei der Übersetzung für M_{ij} die Prämissen $\mathcal{P}$ erstellt wurden, mit $(\{N_{ij} \Downarrow M'_{ij}\} \cup \{n_l \Downarrow m'_l\}_{l\in I_c}) \subseteq \mathcal{P}$, und für alle $M \Downarrow N \in \mathcal{P}$ gilt $s(M) \Downarrow s(N)$, dann gilt $s(M'_{ij}) = s(M_{ij})$.

Beweis: Da $s(M'_{ij}) = s(c(n_1,...,n_a)) = c(s(n_1),...,s(n_a))$ gilt, sind alle $s(n_l)$ für $l \in I_c$ Werte, sonst wäre $s(M'_{ij})$ kein Wert und somit $s(N_{ij}) \Downarrow s(M'_{ij})$ nicht erfüllt. Daher folgt $s(n_l) = s(m'_l)$ aus $s(n_l) \Downarrow s(m'_l)$ und gemäß Induktionsvoraussetzung gilt $s(m'_l) = s(m_l)$. Aufgrund des Übersetzungsverfahrens gilt zudem $n_k = m_k$ für alle $k \in \{1...a\} \setminus I_c$. Es gilt also:

$s(M'_{ij}) = s(c(n_1,...,n_a)) = c(s(n_1),...,s(n_a)) = c(s(m_1),...,s(m_a)) = s(c(m_1,...,m_a)) = s(M_{ij})$

Da wir bereits die benötigten Aussagen für Werte bewiesen haben, können wir die Rückrichtung der Aussage $s \downarrow t \Leftrightarrow s \Downarrow t$ nun zeigen:

Lemma 5.3.24. Sei S ein strukturiertes Reduktionssystem und $R = f(S)$. Falls $s \Downarrow t$ eine Herleitung über R hat, dann gilt auch $s \downarrow t$.

Beweis: Durch Induktion über $\Downarrow$.

Induktionsanfang: Keine Prämissen.

Folgt mit Lemma 5.3.17.

Induktionsvoraussetzung: Angenommen für die Prämissen gilt, falls $s(M_i) \Downarrow s(N_i)$, dann auch $s(M_i) \downarrow s(N_i)$.

Induktionsschritt: z.z.: Falls die Behauptung für die Prämissen gilt, dann gilt sie auch für den Schluss der Regel, d.h. falls $s(M) \Downarrow s(N)$, dann auch $s(M) \downarrow s(N)$:

Fall 1: Der äußere Operator von M ist ein Konstruktor.

Dann wird die folgende Regel angewendet:

$$\frac{\{X_i \Downarrow Y_i\}_{i\in I}}{c(\vec{x}_1.X_1\{\vec{x}_1\},...,\vec{x}_n.X_n\{\vec{x}_n\}) \Downarrow c(\vec{x}_1.Y_1\{\vec{x}_1\},...,\vec{x}_n.Y_n\{\vec{x}_n\})}, \text{ wobei } X_k = Y_k, \text{ für alle } k \neq i.$$

Im GDOS-Format haben wir $|I|$ Ableitungsregeln

$$\left\{\frac{X'_i \to Y'_i}{c(X'_1,...,X'_i,...,X'_n) \to c(X'_1,...,Y'_i,...,X'_n)}\right\}_{i\in I}.$$

Da $s(X_i) \downarrow s(Y_i)$, erhält man durch wiederholtes Anwenden der Ableitungsregeln also auch $c(...s(X_i)...) \downarrow c(...s(Y_i)...)$.

Fall 2: Der äußere Operator von M ist kein Konstruktor.

Dann wird eine Regel der folgenden Form angewendet:

$$r = \frac{\{N_{ij} \Downarrow M'_{ij}\}_{i\in I} \quad \{P_{vj}\}_{v\in I_v} \quad M_j \Downarrow V_j}{F(N_{1j},...,N_{nj}) \Downarrow V_j}.$$

Das Auswertungsaxiom im GDSOS-Format, das zur übersetzten Regel r führt, hat die Form $r' = \frac{}{F(M_{1j},...,M_{nj}) \to M_j}$.

Sei s eine Instanziierung, so dass die Regel r angewendet werden kann, d.h. für alle Prämissen $M_p \Downarrow N_p$ gilt $s(M_p) \Downarrow s(N_p)$.

Gemäß Induktionsvoraussetzung gilt $s(M_j) \downarrow s(V_j)$ und $s(N_{ij}) \downarrow s(M'_{ij})$ für alle $i \in I$, $I = strict(F)$.

Aufgrund der Ableitungsregeln für die strikten Positionen der Form $\frac{X_i \to Y_i}{F(X_1...X_i...X_n) \to F(X_1..Y_i...X_n)}$ gilt:

$F(s(N_{1j}),...,s(N_{nj})) \to^* F(s(M'_{1j}),...,s(M'_{nj}))$, wobei $N_{kj} = M'_{kj}$ für alle $k \in \{1..n\} \setminus I$.

z.z. Es gibt eine Instanziierung s ', die die Anwendung der Regel r' erlaubt, und $s'(M_j) = s(M_j)$ sowie $s'(M_{ij}) = s(M'_{ij})$ für alle $i \in I$ und $s'(M_{kj}) = s(N_{kj})$ für alle $k \in \{1..n\} \setminus I$.

Dies folgt mit Lemma 5.3.23.

Es gilt also:

$F(s(N_{1j}),...,s(N_{nj})) \to^* F(s'(M_{1j}),..., s'(M_{nj})) \to s'(M_j) = s(M_j) \downarrow s(V_j)$ und somit

$F(s(N_{1j}),..., s(N_{nj})) \downarrow s(V_j)$.

Wir haben somit beide Richtungen gezeigt und erhalten das Gesamtergebnis:

Satz 5.3.25. Sei $S = (L, \to)$ ein GDSOS-Reduktionssystem: Dann ist $f(S) = (\varepsilon, R)$ ein strukturiertes Auswertungssystem und für alle s, t aus $T_0(L)$ gilt:

$s \downarrow t$ gdw. $s \Downarrow t$.

Beweis: Folgt mit Lemma 5.3.15., Lemma 5.3.22. und Lemma 5.3.24.

5.4 Darstellung der vorgestellten Kernsprachen und Kalküle

Im Folgenden wollen wir uns damit beschäftigen, wie die bisher vorgestellten Kernsprachen und Kalküle als GDSOS-Reduktionssystem dargestellt werden können. Wir benutzen dazu die in Abschnitt 5.2.2 vorgestellte Variante, die die operationale Semantik durch Reduktionsregeln eines Termersetzungssystems höherer Ordnung (mit Werte-Metavariablen) und der Verwendung von Reduktionskontexten beschreibt und nennen ein solches System von nun an schlicht Reduktionssystem. Zur Betrachtung eines Reduktionssystems genügt die Angabe der GDSOS-Signatur und der Reduktionsregeln, da die Reduktionskontexte gemäß Definition 5.2.6. gegeben sind. Wir werden diese jedoch zwecks Vergleichs mit den Definitionen der Reduktionskontexte in den Kalkülen und Kernsprachen mitbetrachten.

5.4.1 Lambda-Kalkül

Der Lambda-Kalkül ist (abgesehen vom Call-by-Need-Fall, den wir später betrachten) leicht darzustellen:

Call-by-Name:

Die GDSOS-Signatur $L = (O, K, a, strict)$ ist gegeben durch:

$O = \{@, \lambda\}$, $K = \{\lambda\}$, $a(@) = (0, 0)$, $a(\lambda) = (1)$, $strict(@) = \{1\}$ und $strict(\lambda) = \{\}$.

Reduktionskontexte ergeben sich also als $R ::= [] \mid @(R, t)$

Wie bereits in Beispiel 5.1.6. gesehen, ergibt sich die Beta-Reduktion durch die Regel:

$@(\lambda(x.X\{x\}), Y) \rightarrow X\{Y\}$, wobei $a(X) = 1$ und $a(Y) = 0$.

Die GDSOS-Bedingungen sind erfüllt.

Call-by-Value:

Für diesen Fall müssen wir nicht viel ändern, es ist lediglich erforderlich, dass für den Operator @ beide Argumente strikt sind, d.h. $strict(@) = \{1, 2\}$.

Reduktionskontexte sind nun $R ::= [] \mid @(R, t) \mid @(t, R)$.

Wie man sieht, sind die Reduktionskontexte in unseren Reduktionssystemen nicht immer eindeutig, d.h. ein Term kann mehrere Normalordnungsredexe haben. In Reduktionssystemen kann man sich also aussuchen, welche strikten Argumente man zuerst auswertet. Da alle strikten Argumente ausgewertet werden müssen, ist klar, dass die Auswertung im Sinne von $\downarrow$ dadurch nicht beeinflusst wird. Wenn wir für unsere obigen Reduktionskontexte festlegen, dass die strikten Argumente auf Position 1 von @ zuerst ausgewertet werden, ergibt sich die gleiche Wirkung wie durch unsere Definition

$$R_v ::= [] \mid (\lambda x.t)\ R_v \mid R_v\ t$$

aus Abschnitt 3.3.4.

Gemäß GDSOS-Bedingung muss die Metavariable Y in $@(\lambda(x.X\{x\}), Y) \rightarrow X\{Y\}$ nun eine Werte-Metavariable sein, dies drückt genau unseren Wunsch aus, dass ein Term der Form $r \equiv @(\lambda(x.s), t)$ nur dann beta-reduziert werden darf, wenn t ein Wert ist.

5.4.2 KFP

Seien im Folgenden die Konstruktoren der Typen indiziert:

Die GDSOS-Signatur $L = (O, K, a, strict)$ ist gegeben durch:

$$O = K \cup \{@\} \cup \{\texttt{case}_{Typ}\}_{\text{für alle Typen}},$$

$K = \{\lambda\} \cup \{c\}_{\text{für alle Konstruktoren}}$,

$a(@) = (0, 0), strict(@) = \{1\}$,

$a(\lambda) = (1), \quad strict(\lambda) = \{\}$.

Für alle Typen: $a(\texttt{case}_{Typ}) = (0, k_1, ..., k_n)$, *Typ* hat n Konstruktoren mit den Stelligkeiten $k_1, ..., k_n$. $strict(\texttt{case}_{Typ}) = \{1\}$.

Für alle Konstruktoren c: $a(c) = (0, ..., 0)$, so dass $|a(c)| = ar(c)$. $strict(c) = \{\}$.

Reduktionskontexte sind somit: $R ::= [] \mid @(R, t) \mid \texttt{case}_{Typ}(R, \vec{x}_1.t_1, ..., \vec{x}_n.t_n)$

Reduktionsregeln:

$@(\lambda(x.X\{x\}), Y) \rightarrow X\{Y\}$

und für jeden Typ mit den Konstruktoren $c_1, ..., c_n$ und für alle i, mit $1 \le i \le n$:

$$\texttt{case}_{Typ}(c_i(t_1,...,t_{k_i}), \vec{x}_1.X_1\{\vec{x}_1\}, ..., x_1....x_{k_i}.X_i\{x_1,....,x_{k_i}\}, ..., \vec{x}_n.X_n\{\vec{x}_n\}) \rightarrow X_i\{t_1 / x_1,....,t_{k_i} / x_{k_i}\}$$

Die GDSOS-Bedingungen sind erfüllt. Wie man sieht, kann man die Bindung der Case-Alternativen darstellen, allerdings kann man dabei keine Pattern darstellen, die in beliebiger Reihenfolge stehen dürfen, da der $\texttt{case}_{Typ}$-Operator sonst keine feste Stelligkeit hat (es sei denn, man würde auf die Bindung durch die abstrahierten Variablen verzichten, so dass alle Operanden die Stelligkeit 0 haben). Die Case-Alternativen werden also immer in fester Reihenfolge behandelt.

Man kann statt den vielen Case-Ausdrücken für jeden Typen einen einzelnen Case-Ausdruck für alle Typen benutzen, indem man diesen speziellen Case-Ausdruck so definiert, dass dieser alle Case-Alternativen für alle Typen enthält. Für die Betrachtung eines einzelnen Typs kann man dann die anderen Positionen mit Dummy-Argumenten versorgen, die ggf. eine nichtterminierende Berechnung starten – was unserer Konvention, dass wir Typfehler als Nichtterminierung behandeln wollen, entspricht.

5.4.3 KFP+

Rekursive Kombinatoren kann man problemlos definieren, allerdings hat es keinen Sinn, eine allgemeine SK-Beta-Reduktion der Form

$$@(x_1..x_n.X\{x_1,..., x_n\}, t_1,..., t_n) \rightarrow X\{t_1 / x_1,..., t_n / x_n\}$$

zu definieren, da aufgrund der festen Stelligkeiten für Operatoren kein Currying möglich ist. Um Funktionsdefinitionen der Form $F(X_1, ..., X_n) = M$ zu nutzen, ist es daher am einfachsten, diese anhand des Operators λ zu kapseln, also etwa so:

$F(X_1, ..., X_n) = M$

$F'() = \lambda(x_1.\lambda(x_2. \ldots\lambda(x_n.F(x_1, \ldots, x_n))\ldots))$.

5.4.4 PCF

Unser Reduktionssystem ist ungetypt, somit ist das PCF-Typsystem nicht darstellbar. Insbesondere können wir bei Abstraktionen keinen Typ für die Variablen festlegen. Die Berechnung wohlgetypter Ausdrücke lässt sich natürlich nachvollziehen:

GDSOS-Signatur:

$O = K \cup \{@, \mu, \texttt{pred}, \texttt{succ}, \texttt{zero?}, \texttt{ifthenelse}\}$,

$K = \{\lambda, \texttt{True}, \texttt{False}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$

$a(@) = (0, 0)$, $strict(@) = \{1\}$,
$a(\mu) = (1)$, $strict(\mu) = \{\}$,
$a(\texttt{pred}) = (0)$, $strict(\texttt{pred}) = \{1\}$,
$a(\texttt{succ}) = (0)$, $strict(\texttt{succ}) = \{1\}$,
$a(\texttt{zero?}) = (0)$, $strict(\texttt{zero?}) = \{1\}$,
$a(\texttt{ifthenelse}) = (0, 0, 0)$, $strict(\texttt{ifthenelse}) = \{1\}$,
$a(\lambda) = (1)$, $strict(\lambda) = \{\}$,
$a(\texttt{True}) = ()$, $strict(\texttt{True}) = \{\}$,
$a(\texttt{False}) = ()$, $strict(\texttt{False}) = \{\}$.

Für alle natürlichen Zahlen n: $a(n) = ()$, $strict(n) = \{\}$.

Wir nehmen an, dass die natürlichen Zahlen durch unendlich viele Konstruktoren dargestellt werden. Praktisch könnte man zur Darstellung der natürlichen Zahlen Peano-Zahlen verwenden (z.B. Konstruktoren 0 und S, so dass 1 = S(0), 2 = S(S(0), usw.) oder einen internen Datentypen wie Integer benutzen.

Reduktionskontexte sind also:

$R ::= [] \mid @(R, t) \mid \texttt{ifthenelse}(R, s, t) \mid \texttt{pred}(R) \mid \texttt{succ}(R) \mid \texttt{zero?}(R)$

Reduktionsregeln:

$\texttt{ifthenelse}(\texttt{True}, X, Y)$	$\rightarrow X$
$\texttt{ifthenelse}(\texttt{False}, X, Y)$	$\rightarrow Y$
$\texttt{pred}(n)$	$\rightarrow n - 1$, falls $n > 0$
$\texttt{pred}(\texttt{0})$	$\rightarrow \texttt{0}$
$\texttt{succ}(n)$	$\rightarrow n + 1$
$\texttt{zero?}(n)$	$\rightarrow \texttt{False}$, falls $n > 0$
$\texttt{zero?}(\texttt{0})$	$\rightarrow \texttt{True}$
$@(\lambda(x.X\{x\}), Y)$	$\rightarrow X\{Y / x\}$

$$\mu(x.X\{x\}) \qquad \rightarrow X\{\mu(x.X\{x\}) / x\}$$

Wir lassen hier zwecks besserer Übersicht die Klammern bei Termen mit Operatoren der Stelligkeit () weg, eigentlich müsste man z.B. `True`() schreiben.

Die Regeln für `pred`, `succ` und `zero?` lassen sich in der Theorie durch unendlich viele Regeln (also für jeden Konstruktor einer natürlichen Zahl eine) darstellen, was nach den GDSOS-Bedingungen erlaubt ist. Für die Praxis kann man das Regelformat so erweitern, dass man für jeden Operator bei Bedarf eine zusätzliche Regel einbauen darf, die GDSOS-Bedingung (3) verletzen darf, indem diese (bzw. deren linke Seite) die anderen Regeln überlappt. Diese Regel darf dann nur angewendet werden, falls keine andere Regel angewendet werden kann, dies kann man als Vereinigung der restlichen Regeln ansehen.

5.4.5 Call-by-Need-Lambda-Kalkül

Der Call-by-Need-Lambda-Kalkül kann nicht dargestellt werden, da keine Kontexte innerhalb der Reduktionsregeln verwendet werden dürfen. Außerdem werden die Kontexte dazu verwendet, um Abstraktionen zu analysieren. Zudem ist aufgrund der möglichen let-Hüllen der Antworten keine klare Trennung zwischen kanonischen und nichtkanonischen Operatoren möglich. Auch die Definition der Reduktionskontexte weicht stark von Definition 5.2.6. ab. Wir werden im Rahmen der Implementierung unseres Softwaresystems eine Erweiterung der Reduktionssysteme zur Angabe von Kontexten (insbesondere von Reduktionskontexten) und deren Verwendung innerhalb der Regeln betrachten. Unter Verzicht auf die Einhaltung der GDSOS-Bedingungen (so dass wir die GDSOS-Eigenschaften nicht mehr folgern können) kann so die Reduktion des Call-by-Need-Lambda-Kalküls beschrieben werden.

6 Ein Interpreter für Reduktionssysteme

Im Folgenden soll ein Software-System entwickelt werden, das die Spezifikation eines Reduktionssystems einlesen und darauf basierend Terme reduzieren kann. Für die Implementierung benutzen wir die in Kapitel 2 vorgestellte Programmiersprache Haskell.

6.1 Funktionalität der Software

Wir müssen uns zunächst überlegen, welche Funktionen unser System nun genau zur Verfügung stellen soll. Außerdem wollen wir diese Wünsche an die Funktionalität auch gleich auf ihre Realisierbarkeit hin überprüfen. Wir untersuchen dabei erst einfache Möglichkeiten, die dann durch gewünschte Erweiterungen komplexer werden.

Unsere minimalen Anforderungen sind die Fähigkeiten, (1) ein Reduktionssystem einlesen zu können und (2) dieses zu interpretieren, also die beschriebene Reduktion zu vollführen. Somit haben wir schon mal zwei Programmteile: 1. einen Parser und 2. den Interpreter.

Betrachten wir zunächst den Parser: Welche Form soll die Beschreibung eines Reduktionssystems haben?

1. Ansatz: Angabe der GDSOS-Signatur und Definition der Regeln

Dieser Ansatz führt jedoch zu sehr umfangreichen Spezifikationen, insbesondere wenn man noch die Stelligkeiten der Metavariablen mit angibt.

Vereinfachungsmöglichkeiten:

- Eine Regel für einen Operator spezifiziert auch eindeutig dessen Stelligkeit, wenn Metavariablen mit Stelligkeit > 0 wie beschrieben stets in der Form $x_1,...,x_n.X\{x_1,...,x_n\}$ angegeben werden.

- Für nichtstrikte Sprachen ist die Angabe von strikten Positionen nicht notwendig, denn diese ergeben sich aus den Regeln eindeutig dadurch, dass an entsprechenden Positionen keine Metavariablen verwendet werden, sondern Metaterme der Form $c(M_1,...,M_n)$.

2. Ansatz: Die GDSOS-Signatur soll automatisch berechnet werden. Angegeben werden nur die Reduktionsregeln und zusätzlich:

- eine Striktheitsangabe für alle Positionen, die strikt sein sollen, was aber nicht in den Regeln erkennbar ist, da kein expliziter Konstruktor-Metaterm gefordert wird, weil nur Werte-Metavariablen verwendet werden (welche ohne diese Striktheitsangabe nicht als solche erkennbar wären).

- für jeden Konstruktor eine Regel der Form $c(M_1,...,M_n)$ = `constructor`.

Der Parser kann somit wieder in zwei Teile unterteilt werden: (1) eine Einlese-Prozedur des Skriptes in eine Zwischenstruktur und (2) die Analyse der Zwischenstruktur und daraus folgende Berechnung bzw. Zusammenstellung der Reduktionssystem-Datenstruktur.

Wir stellen die Syntax der Regeln zunächst durch eine kontextfreie Grammatik dar und können deshalb für die Einlese-Prozedur den Parsergenerator „Happy“ (siehe [MG01], wir werden später noch näher darauf eingehen) verwenden. Die Berechnung der Stelligkeiten und Striktheitsdefinitionen erfordert lediglich ein Durchlaufen aller Regeln und sollte uns somit auch keine größeren Probleme bereiten.

Widmen wir uns daher dem Interpreter:

Wir gehen nun davon aus, das uns eine Datenstruktur zur Verfügung steht, die uns alle gewünschten Informationen über das Reduktionssystem liefern kann, z.B. auch die Menge aller Reduktionsregeln, die für einen bestimmten Operator definiert sind. Um einen gegebenen Term zu reduzieren, müssen wir überprüfen, ob alle strikten Argumente ausgewertet sind und falls nein, zunächst diese reduzieren. Ansonsten müssen wir die richtige Reduktionsregel suchen (die zu dem gegebenen Term instanziiert werden kann) und erhalten dann das Redukt als Instanz der rechten Seite der Reduktionsregel.

Wunsch: Sharing soll verwendet werden, wenn eine Regel einen Ausdruck kopieren will.

Eine relativ einfache Möglichkeit, diesen Wunsch zu realisieren, bietet die Verwendung einer Template Instantiation Machine. Im Vergleich zu der Standard-Implementierung (wie in [PJL91]) sind ein paar Änderungen nötig: Unter anderem ist kein Currying möglich, so dass eine Reduktion eines Terms $F(t_1,...,t_n)$ nur als ganzes möglich ist. Im Großen und Ganzen können wir das Konzept, das wir später etwas ausführlicher betrachten, aber übernehmen.

Wir könnten mit unserer bisherigen Spezifikation also die im letzten Kapitel dargestellten Reduktionssysteme für den Lambda-Kalkül, KFP und PCF verarbeiten. Wir würden aber auch gerne den Call-by-Need-Lambda-Kalkül verwenden. Dazu müssen wir unsere Definition der Metaterme erweitern, so dass diese auch Kontexte verwenden dürfen. Als Operanden auf der linken Seite einer Regel erlauben wir dann zusätzlich Metaterme der Form $C[X]$ und $x.C[x]$, wobei C ein Kontext ist, X eine Metavariable und x eine Variable, und auf der rechten Seite erlauben wir eine beliebige Verwendung eines Unterterms $C[t]$, wenn C auf der linken Seite benutzt wurde. Für den Kontext C muss jetzt noch eine eigene Definition angegeben werden, diese soll die Form $C = [] \mid F(...C...) \mid ...$ haben. Eine Regel, die $C[X]$ oder $x.C[x]$ enthält, gilt dann für alle Kontexte, die dieser Definition entsprechen. Neben der Verwendung von Kontexten in den Regeln soll außerdem noch die Möglichkeit bestehen, den Reduktionskontext zu spezifizieren.

Für den Interpreter bedeutet dies eine große Veränderung, denn es können nicht mehr einfach die strikten Argumente ausgewertet werden, sondern es muss genau geprüft werden, wo sich gemäß Definition des Reduktionskontextes der entsprechende Redex befindet. Außerdem sind nun Regeln, die ein $C[X]$ oder $x.C[x]$ enthalten, schwieriger zu instanziieren, da auf der rechten Seite für $C[t]$ ja genau der Kontext der linken Seite verwendet werden soll. Somit darf dieser Kontext auch nur einmal auf der linken Seite vorkommen, da sonst nicht klar ist, welche Version auf der rechten Seite zu verwenden ist. Wir erlauben daher Kopien von Kontext-Definitionen anzufertigen, etwa $C' ::= C$. Will man einen Kontext zweimal auf der linken Seite verwenden, kann man also den Kontext und eine Kopie dieses Kontextes verwenden. Es ist klar, dass dieses auch zu ungültigen Definitionen führen kann z.B. durch $D ::= D'$ und $D' :: = D$, was den Programmablauf in

eine Endlosschleife führt, also zu einem Programmabsturz führt. Man könnte dies durch eine Domain-Analyse der Kontext-Definitionen verhindern, dies würde jedoch den Rahmen dieser Arbeit sprengen, so dass der Benutzer selbst darauf achten muss, dass er keine solchen Definitionen benutzt.

Für die Verwendung einer Regel kann der passende Kontext nun durch rekursive Suche über die Struktur gefunden werden. Im Normalfall, also ohne verschachtelte, sich gegenseitig benutzende Definitionen, terminiert die Suche, da man jederzeit überprüfen kann, ob der äußere Operator des jeweiligen Unterterms der Richtige ist und Terme nur endlich viele Unterterme haben. Ist die passende „Instanz“ einer Kontextdefinition gefunden, muss eine passende Struktur angelegt werden, die diese Instanz enthält, damit der Kontext auf der rechten Seite benutzt werden kann. Die Details dieses Verfahrens werden wir bei der Implementierung sehen.

Neben Kontexten erlauben wir noch die Definition syntaktischer Mengen, wie z.B. die Antworten im Call-by-Need-Lambda-Kalkül. Die Verfahrensweise ist dann wie bei Kontexten, nur dass wir uns nicht die Lochpositionen merken müssen. Im Folgenden benutzen wir für diese syntaktischen Mengen den Begriff *syntaktische Domänen* oder einfach das englische Wort *Domains*. Bei der Definition der Domains kann angegeben werden, ob die in der syntaktischen Domäne enthaltenen Terme auch als Werte zählen sollen. Zur Überprüfung, ob ein Term einen Wert darstellt, reicht es also nicht mehr, die Konstruktoren zu betrachten, es müssen auch die Domains geprüft werden.

Für die Fälle, in denen der Standard-Reduktionskontext, wie in Definition 5.2.6. beschrieben, verwendet werden soll, soll dieser natürlich auch automatisch berechnet werden können. Für die Spezialfälle, in denen ein Reduktionskontext nur leicht vom Standard-Reduktionskontext abweichen soll, erlauben wir auch zwecks Vereinfachung eine Umdefinierung dieses Standard-Reduktionskontextes für einzelne Operatoren.

Neben der Verwendung als eigenständiges Programm soll unser System auch zahlreiche Schnittstellen für andere Programme bieten; ein sinnvoller Einsatz unserer Reduktionsmechanismen wäre z.B. im Rahmen einer Gleichheitsanalyse wie in [Man99] beschrieben denkbar. Für diesen Zweck bietet es sich auch an, Nichtdeterminismus mitzubetrachten: Für die Reduktion innerhalb unseres Programms hat dies nicht so viel Sinn, da ja bei mehreren Möglichkeiten nicht klar ist, mit welcher denn nun fortzufahren ist. Aber im Rahmen einer Schnittstelle müssen wir nur statt eines einzelnen Ergebnisses eines Reduktionsschrittes die Menge (bzw. Liste) aller Ergebnisse zurückliefern, das aufrufende Programm muss dann selbst entscheiden, mit welcher Möglichkeit fortzufahren ist. Treffen wir mit unserem Interpreter auf eine nichtdeterministische Regel, so brechen wir die Auswertung mit entsprechendem Hinweis ab. Innerhalb der Reduktionssystem-Definition fordern wir, dass die nichtdeterministischen Regeln gekennzeichnet werden. Nicht gekennzeichnete Regeln werden einfach der Reihe nach überprüft und die erste passende Regel wird dann genommen. Als dritte Möglichkeit bieten wir noch an, sicherzustellen, dass eine Regel nicht mit den anderen überlappt. Für Kontexte bieten wir ebenfalls die Möglichkeit an, die erste passende Alternative zu nehmen oder eben nichtdeterministisch alle Alternativen zu prüfen, wobei der Interpreter aber dann auch abbricht.

Bevor der Interpreter mit der Reduktion beginnt, wollen wir bei Bedarf auch überprüfen können, ob ein Reduktionssystem sich an die GDSOS-Bedingungen hält oder nicht. Bei Verstößen soll natürlich auch darauf hingewiesen werden, worin dieser Verstoß besteht. Die Überprüfung der Bedingungen ist im Wesentlichen durch ein Durchlaufen der Metaterme einer Regel möglich, lediglich für die Bedingung der Nichtüberlappung müssen noch die Regeln untereinander geprüft werden. Wenn wir uns vorher davon überzeugt haben, das die Regeln linear sind, so ist aber auch diese Überprüfung leicht durchführbar. Wir müssen uns ansonsten lediglich beim Laden merken, ob in der Definition eines Reduktionssystems bzgl. GDSOS-Bedingungen ungültige Erweiterungen benutzt wurden.

6.2 Implementierung

Es sollen nun die Funktionsweisen der einzelnen Programmteile dargestellt werden, wobei wir nur die besonders interessanten Code-Fragmente genauer betrachten wollen. Die Funktionsweisen von Programmteilen, deren Entwicklung weniger interessant ist oder deren Code-Umfang, gemessen an der Funktionalität, unverhältnismäßig lang ist, werden im Folgenden also zusammengefasst erklärt. Einige Datenstrukturen von eher allgemeinem Interesse, deren Erläuterung innerhalb der nachfolgenden Betrachtungen nur ablenken würde, werden dagegen im Anhang kurz vorgestellt. Das vollständige Programm mit dem Namen *IfRS* (Interpreter for Reduction Systems) ist im Internet über die Adresse http://www.ki.informatik.uni-frankfurt.de/ zu finden.

Gemäß unserer Vorüberlegungen gliedert sich das Programm zunächst in drei Teile, und zwar die Lade-Prozedur, die Analyse des Reduktionssystems bzw. die Überprüfung der GDSOS-Bedingungen und den eigentlichen Interpreter. Hinzu kommen noch die Datenstrukturen, die unser Reduktionssystem und zugehörige Daten speichern und von jedem der drei Programmteile benötigt werden und somit zuerst entwickelt werden müssen.

6.2.1 Datenstrukturen für ein Reduktionssystem

Wie kann also nun ein Reduktionssystem intern repräsentiert werden? Ein Reduktionssystem enthält globale Einstellungen, wozu wir unter anderem auch den Reduktionskontext zählen, die Definitionen der Domains und Kontexte, sowie die Operatordefinitionen, die auch die Reduktionsregeln enthalten. Auf einfache Weise mit einem Datentyp ausgedrückt sieht das also wie folgt aus:

```
data RedSysP = RedSysP {
  rsGlobalP :: RsGlobalP,
  rsContextsP :: [ContextP],
  rsDomainsP :: [DomainP],
  rsOpDefsP :: [OpDefP]
  } deriving (Show,Eq)
```

Das angehängte „P“ in den verwendeten Namen soll ausdrücken, dass wir in den Typen `RsGlobalP`, `ContextP`, `DomainP` und `OpDefP` auch die Position der zugehörigen Bezeichner in der eingelesenen Reduktionssystem-Definition speichern wollen, um so gegebenenfalls auf die entsprechenden Positionen verweisen zu können. Wir werden später eine Typklasse verwenden, um im Programm zwischen mehreren unterschiedlichen

Repräsentationen wechseln zu können, aber zunächst betrachten wir diese einfache Repräsentation mit einem Datentypen weiter.

Globale Einstellungen

Die globalen Einstellungen enthalten neben dem Reduktionskontext auch einen Programmkontext und spezielle Schalter. Als Schalter erlauben wir hier auch, die Verwendung von Sharing an- und abzuschalten:

```
newtype RsGlobalSw = RsGlSwSharing Bool
 deriving (Eq)
```

Weitere Schalter benötigen wir zur Zeit nicht, planen im Datentyp aber bereits eine Liste von Schaltern ein, um zukünftig bei Bedarf leicht weitere Schalter einfügen zu können. Beim Reduktionskontext speichern wir auch gleich ein Flag mit, das angibt, ob dieser Reduktionskontext dem Standard-Reduktionskontext entspricht oder nicht. Der Programmkontext ist eigentlich ein normaler Kontext, der nur verwendet wird, wenn dies explizit in den Reduktionsregeln angegeben ist. Was ihn jedoch auszeichnet, ist, dass auch dieser automatisch berechnet werden kann. Somit speichern wir auch hier ein Flag, ob es sich um den Standard-Kontext handelt. Der Datentyp für globale Einstellungen sieht somit wie folgt aus:

```
data RsGlobalP = RsGlobalP {
  rsRedContextP :: (Bool,ContextP),
  rsPrgContextP :: (Bool,ContextP),
  rsGlobalSw :: [RsGlobalSw]
  } deriving (Show,Eq)
```

Kontexte

Kontexte bestehen aus dem Namen, den Kontext-Alternativen, also speziellen Termen, die ein Loch enthalten dürfen und die wir im folgenden Kontextterme nennen, und außerdem den Kontexteinstellungen. Als Einstellung erlauben wir zunächst die besprochenen Möglichkeiten, beim Vergleich mit einem Term, die erste passende Alternative zu nehmen oder nichtdeterministisch alle Alternativen zu prüfen:

```
data CtSwitch =
        CtSwMatchAlt
      | CtSwMatchND
  deriving (Eq)
```

Auch hier sehen wir jedoch eine Liste von Einstellungen vor, um zukünftig weitere Schalter einfügen zu können. Der Datentyp für Kontexte sieht also wie folgt aus:

```
data ContextP = ContextP {
  ctNameElP :: ContextNameP,
  ctTermsP :: [ContextTermP],
  ctOptionsP :: [CtSwitch]
} deriving (Eq)
```

Der Kontextname besteht einfach aus dem Namen und der Position des entsprechenden Bezeichners in der Eingabe:

```
type ContextNameP = PrsElement String
```

Der Typ `PrsElement` erlaubt es uns, beliebige Elemente mit einer Position versehen zu können:

```
data PrsElement a = PrsElement {
  prsElement::a,
  prsElPos::PrsPosition
} deriving (Show)
```

Als Instanz von Eq betrachten wir hier nur die eigentlichen Elemente, die Positionen sollen also bei Vergleichen egal sein:

```
instance (Eq a) => Eq (PrsElement a) where
  (==) x y = ((prsElement x) == (prsElement y))
```

Eine Positionsangabe besteht aus dem Dateinamen, sowie der Zeile und Spalte:

```
data PrsPosition = PrsPosition {
  prsPosFile :: PrsFileName,
  prsPosLine :: PrsLineNumber,
  prsPosCol  :: PrsColNumber
} deriving (Eq,Show)

type PrsFileName = FilePath -- String-Synonym aus IO-Modul
type PrsLineNumber = Int
type PrsColNumber = Int
```

Als Trick, um Positionen in anderen Datentypen zu verwenden, deren Instanz von Eq automatisch abgeleitet werden soll, aber ohne dass die Positionen mitberücksichtigt werden, definieren wir eine Variante, bei der alle Positionen gleich sind:

```
newtype PtPosition = PtPosition {
  ptPos::PrsPosition
} deriving (Show)

instance Eq PtPosition where
  (==) a b = True
```

Widmen wir uns nun den Kontexttermen: Wie bereits erwähnt, handelt es sich hier um Terme, die ein Loch enthalten. Außerdem sind Verweise zu anderen Kontexten und Domänen möglich. Aus der Sicht des Datentyps ist ein Kontextterm also entweder

- ein Loch,
- eine Variable,
- ein Verweis auf einen anderen Kontext,

- die Einsetzung eines Kontextterms in einen Kontext,
- der Verweis auf eine syntaktische Domäne,
- eine Abstraktion, die Variablen eines Kontextterms bindet,
- oder die Anwendung eines Operators auf seine Operanden, die allesamt Kontextterme sind.

Der Datentyp sieht somit wie folgt aus:

```
data ContextTermP   =
      CtEmptyP CtPosition
    | CtVarP ContextVariableP
    | CtRefContextP ContextNameP
    | CtUseContextP ContextNameP ContextTermP
    | CtDomainP DomainNameP
    | CtAbstractionP [ContextVariableP] ContextTermP
    | CtOperatorP OperatorP [ContextTermP]
  deriving (Eq)

type CtPosition = PtPosition

type ContextVariableP = PrsElement String

type ContextNameP = PrsElement String

type DomainNameP = PrsElement String
```

Dieser Datentyp könnte allerdings auch mehrere Löcher enthalten oder gar keines. Wir müssen also beim Laden der Kontextdefinitionen darauf achten, dass in jedem Kontextterm genau ein Loch verwendet wird. Ebenso müssen wir sicherstellen, dass Abstraktionen nur als Operanden verwendet werden.

Domains

Eine Domain-Definition ist einer Kontext-Definition sehr ähnlich, der Unterschied ist lediglich, dass ein zusätzliches Flag angibt, ob die enthaltenen Terme als Werte gelten sollen oder nicht (sofern nicht woanders als Wert definiert),

```
data DomainP = DomainP {
  domNameElP :: DomainNameP,
  domTermsP :: [DomainTermP],
  domIsValueP :: Bool,
  domOptionsP :: [DomSwitch]
} deriving (Eq)
```

außerdem enthalten Domain-Terme natürlich keine Löcher:

```
data DomainTermP =
      DomVarP DomainVariableP
    | DomRefDomainP DomainNameP
    | DomUseContextP ContextNameP DomainTermP
    | DomAbstractionP [DomainVariableP] DomainTermP
    | DomOperatorP OperatorP [DomainTermP]
  deriving (Eq)

type DomainVariableP = PrsElement String
```

Besondere Einstellungen werden zunächst nicht benötigt, aber um zukünftig leicht welche hinzufügen zu können, benutzen wir einen Platzhalter:

```
type DomSwitch = String
```

Operanden und ihre Definitionen

Was jetzt noch fehlt, ist der Datentyp für Operatoren, außerdem fehlt ja vom Reduktionssystems-Datentyp auch noch der Datentyp für Operator-Definitionen. Eine Operator-Definition enthält dabei einfach den Operator und alle Regeln für diesen Operator, außerdem planen wir wie gewohnt spezielle Einstellungen ein, auch wenn wir hier zur Zeit keine benötigen und somit nur einen Platzhalter verwenden:

```
type OdSwitch = String

data OpDefP = OpDefP {
 odGetOpP :: OperatorP,
 odGetSwP :: [OdSwitch],
 odGetRulesP :: [RuleP]
 } deriving (Show,Eq)
```

Für einen Operator soll jederzeit

- der Name,
- die Stelligkeit,
- die Striktheits-Information
- und eine Information, ob der Operator kanonisch bzw. ein Konstruktor ist,

erhalten werden können. Die Stelligkeit besteht dabei einfach aus einer Liste der Stelligkeiten der Operanden bzw. der Anzahl der gebundenen Variablen für jeden Operanden. Die Striktheits-Information ist eine Liste der strikten Positionen, d.h. falls i in der Liste enthalten ist, dann ist der i-te Parameter strikt. Zusätzlich zu diesen für das Reduktionssystem relevanten Eigenschaften merken wir uns auch noch die Position des Operatornamens in der Eingabe, so dass unsere Datenstruktur wie folgt aussieht:

```
data OperatorP = OperatorP {
  opNameP :: String,
  opPositionP :: PrsPosition,
  opArityP :: [OpArityNum],
  opCanonicalP :: Bool,
  opStrictP :: [OpStrictNum]
} deriving (Show)

type OpArityNum = Int

type OpStrictNum = Int
```

Regeln

Eine Regel besteht nun aus einem Operator, einer Liste von Metatermen (als Operanden) und einem Metaterm (als Ergebnis) sowie den Regel-Optionen:

```
data RuleP = RuleP {
     ruleOpP :: OperatorP,
     ruleSwP :: [RuleSwitch],
     ruleArgsP :: [MetatermP],
     ruleResultP :: MetatermP
} deriving (Eq)
```

Die Regel-Optionen bzw. Schalter sind dabei:

```
data RuleSwitch =
        RuleSwEvalAlt
      | RuleSwEvalExcl
      | RuleSwEvalND
  deriving (Eq)
```

Wie in den Vorüberlegungen beschrieben, kann also eine Regel so eingestellt werden, dass die erste passende Alternative genommen wird, oder als exklusiv definiert werden, so dass sie nicht mit einer anderen Regel überlappen darf, oder aber als nicht-deterministisch definiert werden.

Metaterme

Die Definition der Metaterme ähnelt den Kontext- und Domaintermen, die Besonderheit sind hier die Meta-Variablen und die Meta-Applikation:

```
data MetatermP =
    MtVariableP VariableP
  | MtMetaVarP MetaVarP
  | MtAbstractionP [VariableP] MetatermP
  | MtMetaAppP MetaVarP [MetatermP]
  | MtOperatorP OperatorP [MetatermP]
```

```
  | MtContextP ContextNameP MetatermP
  | MtDomainP DomainNameP
 deriving (Eq)
```

Variablen und Meta-Variablen

Die gewöhnlichen Variablen bestehen einfach aus dem Variablennamen zusammen mit der Position in der Eingabe, so dass wir den Namen durch den Datentyp `PrsElement` kapseln:

```
type VariableP = PrsElement String
```

Die Meta-Variablen verfügen, wie im letzten Kapitel gesehen, gegenüber normalen Variablen noch über eine Stelligkeit und ein Flag, das angibt, ob nur Werte enthalten sein dürfen oder auch beliebige Terme:

```
data MetaVarP = MetaVarP {
     mvarNameP::String,
     mvarPositionP::PrsPosition,
     mvarArityP::Int,
     mvarIsValueP::Bool
} deriving (Show)
```

Somit haben wir jetzt einfache Datenstrukturen beisammen, die alle relevanten Informationen eines Reduktionssystems speichern können. Die Verwendung der Operatoren inkl. Stelligkeit, Striktheits-Information und Konstruktor-Flag in den Meta-, Kontext- und Domaintermen ist allerdings etwas mit Vorsicht zu genießen, denn da diese Informationen ja eigentlich Teil der Operatordefinition sind, sind diese bei wiederholter Verwendung eines Operators redundant, was leicht zu Inkonsistenzen führen kann: Wenn ein Operator wiederholt verwendet wird, muss also sichergestellt werden, dass immer die gleiche Stelligkeit usw. im Operator-Typ gespeichert wird! Außerdem verschwenden Redundanzen natürlich Speicherplatz, was in diesem Fall jedoch nicht so viel ausmacht, falls die Anzahl der Operanden nicht so hoch ist und damit die Stelligkeits-Liste nicht zu lang. Man kann diese Redundanzen leicht vermeiden, indem man in den Metatermen usw. nur die Operatornamen verwendet und bei Funktionen, die die Stelligkeit usw. benötigen, die Liste der Operatordefinitionen als zusätzliches Argument übergibt. Dies hat dann jedoch den Nachteil, dass wir den Operator in den Operatordefinitionen erst suchen müssen. Da wir die entsprechenden Informationen im Analyse-Teil unseres Programms oft benötigen, bleiben wir also aufgrund des Geschwindigkeitsvorteils bei diesem einfachen Ansatz.

Darstellung durch Multiparameter-Klassen

Wie bereits erwähnt, wollen wir aber auch Typklassen verwenden, um leicht zwischen unterschiedlichen Datenstrukturen wechseln bzw. verschiedene Repräsentationen für Reduktionssysteme gleichzeitig im Programm verwenden zu können. Um eine größtmögliche Flexibilität zu wahren, benutzen wir Typklassen für jeden Teil der Hierarchie, d.h. die Reduktionssystem-Typklasse greift auf eine Regel-Klasse zu, die die Metaterm-Klasse benutzt, die wiederum auf die Metavariablen-, Variable- und Operatoren-Klassen zurückgreifen. Für die Darstellung der Kontexte und Domänen brauchen wir

ebenfalls entsprechende Klassen. Wie bei den Operatoren gesehen, kann es dabei sinnvoll sein, Informationen nicht direkt in einem Datentypen zu speichern, sondern zusätzlich ein zweites Argument, das diese benötigten Informationen beinhaltet und das wir im Folgenden als Umgebung bezeichnen, mitzubetrachten. Dies ermöglicht uns dann später ebenfalls, die Adresse eines Speicherplatzes im Zustand der Template Instantiation Machine als Term zu verwenden.

Um sowohl den eigentlichen Datentypen als auch die Umgebung variabel halten zu können, benötigen wir Multiparameter-Klassen. Diese sind eine Erweiterung von Haskell, so dass wir diese Erweiterungen für GHC durch einen Steuerungsbefehl

```
{-# OPTIONS -fglasgow-exts -cpp #-}
```

(in Kommentarform, so dass andere Haskell-Implementierungen dies ignorieren) anschalten müssen, bzw. Hugs durch einen (Kommandozeilen-)Aufruf des Interpreters mit zusätzlichem Parameter `-98` starten müssen. Zu beachten ist nun, dass die Funktionen alle vorgesehenen Typvariablen als Parameter enthalten müssen, es sei denn, man gibt *Abhängigkeiten*[40] an, so dass der Typ eines nicht enthaltenen Parameters aus den anderen Parametern eindeutig hervorgeht. Abhängigkeiten werden einfach nach einem `|` in der Form `a -> b` angegeben, wenn der Typ `b` von `a` abhängig sein soll. Dies benötigen wir auch bei der hierarchischen Gliederung der Klassen, da wir z.B. in der Metaterm-Klasse sowohl enthaltene Operatoren als auch enthaltene Meta-Variablen zurückliefern wollen, aber diese nur als Parameter angeben, wenn sie die entsprechende Funktion wirklich benötigt.

Term-Klasse

Wir wollen nicht alle Typklassen im Detail behandeln, da sich deren Aufbau untereinander sehr ähnelt und diese auch sehr eng dem Schema der eben beschriebenen Datentypen folgen. Exemplarisch betrachten wir somit eine neue Term-Klasse, die dann wie erwähnt im Zusammenhang mit der Template Instantiation Machine benötigt wird; diese trägt der Tatsache Rechnung, dass wir bei Termen nicht die abstrakten Bestandteile der Metaterme, sprich Meta-Variablen, Meta-Applikationen und Kontext- und Domain-Referenzen, benötigen:

```
class (Variable v env, Operator op env) =>
  Term t env v op | t -> v, t -> op where
    trmIsVariable :: t -> env -> Bool
    trmIsAbstraction :: t -> env -> Bool
    trmIsOpTerm :: t -> env -> Bool
    trmGetVariable :: t -> env -> v
    trmGetAbsVars :: t -> env -> [v]
    trmGetAbsTerm :: t -> env -> t
    trmGetOp :: t -> env -> op
    trmGetOpTerms :: t -> env -> [t]
```

[40] engl. *dependencies*, als *functional dependencies* auch abgekürzt *FunDeps*.

Die Term-Klasse benutzt vier Typvariablen, wobei nur die ersten beiden, die Term-Datenstruktur `t` und die Umgebung `env`, ständig benötigt werden. Die Parameter `v` und `op`, bei denen wir zusammen mit der gemeinsamen Umgebung eine Zugehörigkeit zur Variablen- bzw. Operator-Klasse voraussetzen, müssen also abhängig von `t` sein. Da wir ausschließlich die Methoden der Term-Klasse benutzen wollen, können wir keine Fallunterscheidung über die Konstruktoren des verwendeten Datentyps machen und benötigen somit, zusätzlich zu den Selektor-Methoden `trmGetVariable`, `trmGetAbsVars`, `trmGetAbsTerm`, `trmGetOp` und `trmGetOpTerms` auch die Methoden `trmIsVariable`, `trmIsAbstraction` und `trmIsOpTerm`, die uns Informationen über den Term-Inhalt geben und somit zur Fallunterscheidung benutzt werden können.

Klassen für Suchstrukturen

Zusätzlich zu den Klassen, die direkt zu den besprochenen Datentypen korrespondieren, benutzen wir noch zusätzliche Klassen um Suchstrukturen zu ermöglichen, so definieren wir z.B. neben der OpDef-Klasse noch eine Klasse OpDefs:

```
class (OpDef od env op rs r mt v mv cn dn) =>
  OpDefs ods env od op rs r mt v mv cn dn |
   ods -> od, od -> op, od -> rs, rs -> r,
   r -> op, r -> mt, mt -> v,
   mt -> mv, mt -> op, mt -> cn, mt -> dn where
    odList :: ods -> env -> [od]
    odLookUp :: String -> ods -> env -> Maybe od
```

Die Nachschlage-Funktion können wir dabei wie folgt vordefinieren:

```
    odLookUp n ods env =
      case (filter
             (\x -> ((opName (odGetOp x env) env) == n))
               (odList ods env)) of
        [] -> Nothing
        (x:xs) -> Just x
```

Für spezielle Suchstrukturen (z.B. Hash-Tabellen) sollte man die Nachschlage-Funktion natürlich neu definieren, aber ansonsten reicht die Definition von `odList`, die einfach die Liste aller Operator-Definitionen zurückliefern soll. Da wir mit Haskell-Erweiterungen auch Typsynonyme als Klassen-Instanzen verwenden können, benutzen wir der Einfachheit halber zunächst `[OpDefP]`:

```
instance OpDefs [OpDefP] env OpDefP OperatorP [RuleP] RuleP
MetatermP VariableP
  MetaVarP ContextNameP DomainNameP where
    odList ods env = ods
```

Reduktionssystem-Klasse

Auch sonst reicht es bei den Klassen-Instanziierungen, die Umgebung zu ignorieren und die entsprechenden Selektoren der Datentypen zu benutzen. Betrachten wir dazu abschließend die Reduktionssystems-Klasse:

```
class (RsGlobal rg env c cn cts ct cv dn op,
       Contexts cs env c cn cts ct cv dn op,
       Domains ds env d dn doms dom dv cn op,
       OpDefs ods env od op rs r mt v mv cn dn) =>
  RedSys redsys env rg cs c ds d ods od cn cts ct cv dn doms
   dom dv rs r mt op mv v |
    redsys -> rg, redsys -> cs, redsys -> ds, redsys -> ods,
    rg -> c, c -> cn, c -> cts, cts -> ct, ct -> cv,
    ct -> cn, ct -> dn, ct -> op, cs -> c,
    ds -> d, d -> dn, d -> doms, doms -> dom, dom -> dv,
    dom -> dn, dom -> cn, dom -> op,
    ods -> od, od -> op, od -> rs, rs -> r, r -> op, r -> mt,
    mt -> v, mt -> mv,
    mt -> op, mt -> cn, mt -> dn where
        rsGlobal :: redsys -> env -> rg
        rsContexts :: redsys -> env -> cs
        rsDomains :: redsys -> env -> ds
        rsOpDefs :: redsys -> env -> ods
        rsLookUpContext :: String -> redsys -> env -> Maybe c
        rsLookUpDomain :: String -> redsys -> env -> Maybe d
        rsLookUpOpDef :: String -> redsys -> env -> Maybe od

        rsLookUpContext n rs env =
          let rg = rsGlobal rs env
              (_,rc) = rsGetRedContext rg env
              (_,pc) = rsGetPrgContext rg env
              rcn = contextName (ctGetName rc env) env
              pcn = contextName (ctGetName pc env) env
                in
            case n of
              _ | n == rcn -> (Just rc)
              _ | n == pcn -> (Just pc)
              _ -> ctLookUp n (rsContexts rs env) env

        rsLookUpDomain n rs env =
                     domLookUp n (rsDomains rs env) env

        rsLookUpOpDef n rs env =
                     odLookUp n (rsOpDefs rs env) env
```

Wie man sieht, gestaltet sich die Definition durch die vielen benötigten Unterklassen und dadurch bedingten Abhängigkeiten etwas aufwendig, aber die Methoden sind sehr einfach gehalten, die Nachschlage-Funktionen `rsLookUpContext` (die zusätzlich noch den

Reduktions- und den Programmkontext einbezieht), `rsLookUpDomain` und `rsLookUpOpDef` sind vordefiniert und basieren auf den entsprechenden Nachschlage-Funktionen der Klassen `Contexts`, `Domains` und `OpDefs`. Bei den Instanzen der Klasse RedSys müssen also nur die Selektor-Funktionen `rsGlobal`, `rsContexts`, `rsDomains` und `rsOpDefs` definiert werden:

```
instance RedSys RedSysP env RsGlobalP [ContextP] ContextP
  [DomainP] DomainP [OpDefP] OpDefP ContextNameP
  [ContextTermP] ContextTermP ContextVariableP DomainNameP
  [DomainTermP] DomainTermP DomainVariableP [RuleP] RuleP
  MetatermP OperatorP MetaVarP VariableP where
    rsGlobal rs _ = rsGlobalP rs
    rsContexts rs _ = rsContextsP rs
    rsDomains rs _ = rsDomainsP rs
    rsOpDefs rs _ = rsOpDefsP rs
```

Zu guter Letzt sei noch erwähnt, dass wir folgendes Typsynonym als offiziellen Typnamen benutzen:

```
type IfrsRedSys = RedSysP
```

6.2.2 Einlesen einer Reduktionssystem-Definition

Nachdem wir nun wissen, wie wir das Reduktionssystem intern darstellen können, widmen wir uns also der Aufgabe, eine Reduktionssystem-Definition in diese Datenstruktur einzulesen. Zunächst benötigen wir dazu eine endgültige Festlegung, welches Format eine solche einzulesende Definition haben soll:

6.2.2.1 Die Syntax eines Reduktionssystem-Skriptes

Wir gliedern das Skript in 3 Teile: Zunächst wollen wir eine Möglichkeit bieten, die Definitionen anderer Skripte einzubinden. Dazu erlauben wir Befehle der Form

`include` Dateiname.

Dies hat dann die gleiche Wirkung, wie wenn man die Reduktionsregeln, Kontext-Definitionen (ohne Reduktions- und Programmkontext) und syntaktische Domänen der angegebenen Datei direkt in das aktuelle Skript kopiert.

Danach können die globalen Eigenschaften eingestellt werden: man kann angeben, ob der Interpreter Sharing verwenden soll, durch den Befehl

`sharing on` bzw. `sharing off`

und man kann die Reduktions- und Programmkontexte definieren:

`reductioncontext` Name = Kontext-Alternativen (optional: <Schalter>)

`programcontext` Name = Kontext-Alternativen (optional: <Schalter>).

Die Kontext-Alternativen sind eine Aufzählung von Kontexttermen, jeweils durch ein | getrennt. Anstatt eines Kontexttermes können aber auch zusätzlich die Schlüsselwörter `rstandard` und `cstandard` verwendet werden, für die dann die vorberechneten Reduktions- bzw. Programmkontext-Alternativen eingefügt werden; es kann sogar in Klammern dahinter ein Operatorname angegeben werden, dann werden nur die zu diesem Operator passenden Alternativen eingefügt. Als Schalter erlauben wir `<match_alt>` (deterministisch – die erste passende Alternative wird genommen) oder `<match_nd>` (nichtdeterministisch – alle Möglichkeiten werden geprüft). Falls dieser Schalter nicht angegeben wird, ist `<match_alt>` voreingestellt.

Im Hauptteil des Skriptes können dann weitere Kontexte sowie syntaktische Domänen definiert werden, außerdem werden hier die Reduktionsregeln angeben. Kontext- bzw. Domänen-Definitionen und Reduktionsregeln können sich dabei beliebig abwechseln, es sollen lediglich alle Regeln für einen Operator direkt hintereinander stehen.

Kontext-Definitionen haben die gleiche Form wie die Definition der Reduktions- bzw. Programmkontexte, außer dass hierbei das Schlüsselwort `context` verwendet wird. Syntaktische Domänen haben ebenfalls fast diese Form, nur dass natürlich keine Kontextdefinition angegeben wird, sondern eine spezielle Domänendefinition, die keine Löcher erlaubt. Das Schlüsselwort ist `domain`, mit einem vorangestellten `value` wird diese syntaktische Menge als Teil der Wertemenge festgelegt.

Bei den Regeln unterschieden wir die Definition von Konstruktoren und die Reduktionsregeln für nicht-kanonische Operatoren. In beiden Fällen können zuvor die folgenden Definitionen vorgenommen werden:

Mit

`strict`(Operatorname) = strikte Positionen

können die strikten Argumente definiert werden, diese gelten im voreingestellten Fall zusätzlich zu den berechneten strikten Argumenten (aufgrund der Verwendung von Konstruktoren bzw. Werten in den Reduktionsregeln). Durch ein angehängtes `exclusive` wird eine Fehlermeldung erzwungen, wenn eine berechnete strikte Position nicht in der vom Benutzer angegebenen Definition enthalten ist.

Als weitere Möglichkeiten können mit

`rstandard`(Operatorname) = Kontext-Alternativen

und

`cstandard`(Operatorname) = Kontext-Alternativen

die Standardberechungen des Reduktions- bzw. Programmkontextes für diesen Operator umdefiniert werden. Die Kontextterme dürfen hier aber nur diesen Operator (für den die Standardberechnung umdefiniert werden soll) an äußerer Position enthalten und keinen anderen. Um sicherzustellen, dass nicht andere Kontextterme mit anderem äußeren Operator durch Referenzen implizit hinzugefügt werden, sind auch Referenzen zu anderen

Kontexten als Kontextalternative verboten. Die Kontext-Referenzen können aber weiterhin als Operanden innerhalb eines Kontextterms verwendet werden. Das Schlüsselwort `recursive` kann als Operand dazu verwendet werden, rekursiv zu einer Kontextdefinition, die `rstandard` oder `cstandard` benutzt, zu verweisen, d.h. die hier definierten Kontextterme werden dann beim Laden in die Kontextdefinition eingefügt und die Vorkommen von `recursive` durch einen Verweis auf den eigenen Kontextnamen ersetzt. Das Schlüsselwort `ignore` als alleinige Kontextalternative kann hingegen verwendet werden, um zu verhindern, dass der aktuelle Operator zur Standard-Berechnung beiträgt.

Kommen wir zurück zu den Regeln:

Eine Konstruktor-Definition hat schlicht die Form

Operatorname(Argumente) = `constructor`,

wobei die Argumente einfach Metavariablen (für Metavariablen mit Stelligkeit > 0 in der Form $x_1...x_n.X\{x_1,...,x_n\}$ geschrieben) sind.

Die Reduktionsregeln haben die bereits im letzten Kapitel beschriebene Form, außer dass zusätzlich Kontexte und syntaktische Domänen verwendet werden dürfen. Abstraktionen dürfen dabei auf der linken Seite einer Regel nur in den Formen $x_1...x_n.X\{x_1,...,x_n\}$, x.C[x] und $x_1...x_n.D$ vorkommen, wobei X eine Metavariable, C einen Kontext und D eine Domain bezeichnet. Die Metaterme auf der linken Seite dürfen innerhalb ihrer Operanden keine Kontexte oder Domains enthalten, erlaubt ist eine Verwendung eines Kontexts mit einer Meta-Variablen C[X] oder eine Domain D sonst nur als direkter Operand der Regel bzw. Metaterm der linken Seite.

Nachfolgend kann optional noch ein Schalter für die Regel verwendet werden: `<eval_alt>` ist dabei die Voreinstellung und legt fest, dass beim Suchen der Regel die erste Alternative genommen wird, falls `<eval_excl>` verwendet wird, darf die Regel nicht mit einer anderen überlappen und mit `<eval_nd>` wird die Regel nicht-deterministisch verwendet.

In der Eingabe dürfen außerdem Kommentare im Haskell-Stil verwendet werden.

6.2.2.2 Der Parser

Wie bereits erwähnt, soll das Einlesen des Reduktionssystem-Skriptes aufgeteilt werden in das Parsen der Eingabedatei, wobei das Ergebnis in einer Behelfs-Datenstruktur zwischengespeichert wird, und in eine Untersuchung des Parse-Ergebnisses, bzw. eine Berechnung der Stelligkeiten, strikten Positionen, Reduktionskontexte usw., so dass schließlich das Reduktionssystem mit allen benötigten Informationen zusammengesetzt werden kann.

Während des Parse-Vorganges kann noch nicht zwischen Variablen, Meta-Variablen und Domain-Namen unterschieden werden, ebenso sind Referenzen auf andere Kontexte innerhalb der Kontextdefinitionen auch erst dann als solche erkennbar, wenn alle Kontextdefinitionen eingelesen wurden und man somit nachprüfen kann, ob ein Kontext

mit entsprechendem Namen existiert. Aus diesem Grund werden alle Namen in der Behelfsdatenstruktur zunächst als Identifier gespeichert und die eigentliche Bedeutung wird dann erst später berechnet. Ansonsten folgen die Behelfsdatenstrukturen im Wesentlichen dem Aufbau der in Abschnitt 6.2.1 besprochenen Reduktionssystem-Datentypen.

Lexikalische Analyse

Der Parser gliedert sich nun in einen Lexer, der die Eingabe lexikalisch analysiert und in Token zerlegt, und die Grammatik-Datei für den Parser-Generator Happy. Die Token, die der generierte Parser dann weiterverarbeitet, bestehen dabei aus einem Lexem, d.h. einem Zeichen oder Wort aus der Eingabe, und der Position des Lexems in der Eingabe:

```
type PltToken = PrsElement PltLexem
```

Der Datentyp `PltLexem` hat somit einen Konstruktor für jedes Zeichen und für jedes Schlüsselwort, außerdem für beliebige Namen und einen zusätzlichen Konstruktor, der das Ende der Eingabe markiert; wir verzichten hier aufgrund der vielen Schlüsselwörter auf die genaue Betrachtung der Haskell-Definition. Um uns die Zuordnung der Konstruktoren zu den Schlüsselwörtern zu vereinfachen, benutzen wir eine Assoziationsliste (siehe Anhang), außerdem benutzen wir eine eigene Assoziationsliste für die Spezialzeichen wie z.B. Klammern, da diese ebenso wie Leerzeichen die Schlüsselwörter und Namen voneinander trennen.

Wir definieren zunächst eine Funktion, die aus einem Eingabestring das erste Token extrahiert und dieses zusammen mit dem Reststring ausgibt. Außerdem soll die Position verwaltet werden, dazu muss der Funktion die aktuelle Position übergeben werden und es wird dann die nächste Position zurückgeliefert. Da die Kommentare und Include-Definitionen zu Fehlern führen können, kapseln wir das Ergebnis mit dem Datentyp `ParseResult`, Betrachtungen zur Ausnahmebehandlung siehe Anhang. Der Typ dieser Funktion ist somit:

```
plLex1 :: PrsPosition -> String ->
            ParseResult (PltToken,String,PrsPosition)
```

In der Implementierung dieser Funktion macht man zunächst eine Fallunterscheidung über das erste Zeichen; falls dieses keine besondere Bedeutung hat, wird das gesamte Wort bis zum nächsten Trennzeichen oder Zeilenwechsel betrachtet. Anhand der Assoziationslisten für Schlüsselwörter und Spezialzeichen können wir dann leicht den passenden Lexem-Konstruktor zuordnen. Wir müssen somit nur noch darauf achten, dass wir die Positionen korrekt verwalten und Kommentare und Include-Definitionen die richtige Form haben, d.h. ein mit `{-` eröffneter Kommentar wird auch wieder mit `-}` geschlossen, und der Datei-Name einer Include-Definition wird von Anführungszeichen umschlossen.

Monadischer Parser und „Threaded Lexer“

Da wir den Datentypen `ParseResult` als Ausnahme-Monade, wie in [MG01, Abschnitt 2.5] beschrieben, verwenden wollen, und zwar sowohl im Lexer, als auch im Parser, definieren wir die lexikalische Analyse, die wir für unseren Happy-Parser

verwenden wollen, als „Threaded Lexer“ (vgl. [MG01, Abschnitt 2.5.2]). Der Parser liest nun nicht, wie sonst üblich, eine Liste von Token ein, sondern ruft den Lexer für jedes neue zu lesende Token auf. Dieses Verfahren kann auch zur Kommunikation zwischen Parser und Lexer benutzt werden, wie sie für aufwendigere Layout-Regeln nötig ist – wir benötigen dies jedoch hier nicht. Allerdings müssen wir dafür sorgen, dass der Lexer nun den passenden Teil der Eingabe erhält, ebenso die aktuelle Position. Die Lexer-Funktion muss dabei einen Typen der Form `lexer :: (PltToken -> P a) -> P a` haben, d.h. sie bekommt eine Funktion übergeben, die die weitere Berechnung darstellt[41], der Lexer muss das nächste Token berechnen und dieser Funktion übergeben. Analog zu [MG01, Abschnitt 2.5.3] kann also der Typ

```
type P a = String -> PrsPosition -> ParseResult a
```

dazu verwendet werden, die Eingabe und die aktuelle Position durchzuschleifen. Die Position könnte somit auch während des Parse-Vorganges durch eine Funktion

```
getLineNo :: P PrsPosition
getLineNo = \s p -> (return p)
```

erhalten werden, wobei der Typ `P` dann als Monade (mit noch zu definierenden monadischen Funktionen „then“ and „return“) benutzt wird, sich das `return` in der Definition von `getLineNo` aber auf die Monaden-Instanz des Typen `ParseResult` bezieht. Da wir die Positionsangabe auch noch nach dem Parse-Vorgang benötigen, ist unsere Methode, die Position direkt im Token zu speichern, aber praktischer, denn dadurch dass der von Happy erstellte Parser immer ein Zeichen im Voraus liest, ist eine nachträgliche Speicherung der Position während des Parse-Vorgangs schwieriger zu handhaben.

Wir benutzen aber diese Vorgehensweise, um im Falle eines Parse-Fehlers herausfinden zu können, welches Token zuletzt gelesen wurde. Dazu müssen wir das zuletzt gelesene Token ebenfalls innerhalb der Parse-Monade übergeben. Somit benutzen wir folgenden Typen:

```
type PmFunction a = String -> PrsPosition ->
        PltToken -> ParseResult a
```

Das zuletzt gelesene Token können wir nun mit folgender Funktion erhalten:

```
pmGetToken :: PmFunction PltToken
pmGetToken = \s pos token -> (return token)
```

Den „Threaded Lexer” können wir jetzt analog zu [MG01, Abschnitt 2.5.2] durch eine Wrapper-Funktion unserer Funktion `plLex1` darstellen:

[41] Diese Technik nennt sich „continuation passing style“ (abgekürzt: CPS)

```
plThreadedLexer :: (PltToken -> PmFunction a) -> PmFunction a
plThreadedLexer cont =
  pmThen
    (\s pos _ -> (plLex1 pos s))
    (\(t,xs,newpos) _ _ _ -> cont t xs newpos t)
```

Die von Happy benötigen Monaden-Funktionen „then“ und „return“ definieren wir dabei wie folgt:

```
pmThen :: PmFunction a -> (a -> PmFunction b) -> PmFunction b
m `pmThen` k = \s p t ->
   let prs = m s p t in
     if errResultOk prs then
       let res = errGetResult prs in
         k res s p t
     else
       throwError (errGetInfo prs)

pmReturn :: a -> PmFunction a
pmReturn a = \s p t -> (return a)
```

Außerdem benutzen wir für den Parser noch eine spezielle Funktion, um Fehler zurückzuliefern:

```
pmFail :: PrsErrorInfo -> PmFunction a
pmFail e = \s p t -> (throwError e)
```

Wir haben nun alle Voraussetzung zur Definition unser Happy-Grammatikdatei erfüllt, betrachten wir also diese Definition des Parsers:

Die Grammatik für den Parser-Generator Happy

Zunächst kann mittels der Direktive `%name` der Name der Parse-Funktion spezifiziert werden. Dabei besteht auch die Möglichkeit mehrere Namen für mehrere Einstiegspunkte in die Grammatik zu definieren. Wir wollen zum einen ein komplettes Reduktionssystem parsen können, zum anderen aber auch einfach einen Term. Daher:

```
%name pParseRSFunction InputDef
%name pParseTermFunction TermDef
```

Nun müssen wir den Tokentyp angeben:

```
%tokentype { PltToken }
```

Zur Benutzung der Monade benötigt Happy den Typ und die Namen der Implementierungen für „then“ und „return“ (falls der Typ keine echte Instanz der Monad-Klasse ist):

```
%monad { PmFunction } { pmThen } { pmReturn }
```

Außerdem müssen wir die Lex-Funktion und das Token, welches das Eingabeende spezifiziert, angeben:

```
%lexer { plThreadedLexer } { (PrsElement PltLexEOF _) }
```

Natürlich müssen auch überhaupt alle zu verwendenden Token spezifiziert werden, wir deuten die Definition hier nur an:

```
%token
     '='  { (PrsElement PltLexDef _) }
     ...
     reductioncontext { (PrsElement PltLexRedContext _) }
     ...
     name { (PrsElement (PltLexName _) _) }
     ...
```

Der Inhalt der geschweiften Klammern wird dabei von Happy direkt in entsprechende Pattern Matchings eingesetzt. Durch den Einsatz der Wildcards drücken wir aus, dass diese Teile des Datentyps an dieser Stelle unwichtig sind.

Zwischen den Happy-Deklarationen und der eigentlichen Grammatik muss nun noch ein %% stehen.

Widmen wir uns also der eigentlichen Grammatik:

Eine Produktion besteht aus einem Nicht-Terminal auf der linken Seite, gefolgt von einem `:` und einer oder mehrerer Expansionen, die durch ein `|` getrennt werden. Optional kann vorher, gekennzeichnet durch `::`, noch der Rückgabe-Typ festgelegt werden. Eine Expansion kann aus Terminalen und Nicht-Terminalen bestehen, dahinter steht in geschweiften Klammern der Rückgabe-Wert, wobei durch ein $x der Wert des x-ten (Nicht-)Terminals eingesetzt wird und ansonsten der Haskell-Code unverändert übernommen wird.

Zunächst passiert wenig Spannendes, nämlich einfach eine Auftrennung in Include- und Reduktionssystem-Teil:

```
InputDef :: {PtInputDef}
     : IncludesDef RedSysDef   { (PtInputDef $1 $2) }
```

Wir konzentrieren uns auf den Reduktionssystem-Teil: Dieser teilt sich auf in die globalen Optionen und Definitionen der Kontexte, Domänen oder Regeln:

```
RedSysDef :: {PtRedSysDef}
     : GlobalSwitchesDef CDRsDef { (PtRedSysDef $1 $2) }
```

Bei sämtlichen Listen ist nun zu beachten, dass die von Happy erstellten Parser besser funktionieren, bzw. schneller sind, wenn eine linksrekursive Grammatik verwendet wird. Dies führt dazu, dass wir die Listen zunächst falsch herum konstruieren, da eine Konkatenation (die für die richtige Reihenfolge bei linksrekursiver Grammatik nötig wäre) zu viele Ressourcen braucht (bzw. in der vordefinierten Art und Weise lineare Zeit bzgl.

der Größe der Liste benötigt). Wir müssen daher in der Nachbearbeitung die fertig geparsten Listen mit `reverse` wieder umdrehen. Bis auf diese Besonderheit passiert wenig Überraschendes:

```
GlobalSwitchesDef :: { [PtGlobalSwitchDef] }
     : GlobalSwitchesDef GlobalSwitchDef { $2:$1 }
     | {- Empty -}       { [] }

GlobalSwitchDef :: { PtGlobalSwitchDef }
     : sharing '=' OnOffDef   {PtSwSharing $3}
     | ReductionContextDef    {PtSwRedContext $1 }
     | ProgramContextDef      {PtSwPrgContext $1 }
```

Wir überspringen die Details der globalen Optionen und machen mit den Definitionen der Kontexte, Domänen oder Regeln weiter:

```
CDRsDef :: { [PtCDRDef] }
     : CDRsDef CDRDef    { $2:$1 }
     | {- Empty -}       { [] }

CDRDef :: { PtCDRDef }
     : ContextDef        { PtCtxt $1 }
     | DomainDef         { PtDom $1 }
     | RedDef            { PtRed $1 }
```

Kontext-Definitionen bestehen wie beschrieben aus dem Kontextnamen, den Kontextalternativen und Optionen zum Matching:

```
ContextDef :: { PtContextDef }
     : context NameDef '='
          ContextAltDefs ContextOptionsDef
           { (PtContextDef (plGetPosition $1) $2 $4 $5) }
```

Die Kontext-Alternativen sind, wie beschrieben, eine Aufzählung von Kontexttermen mit zusätzlichen Möglichkeiten durch `rstandard` und `cstandard`:

```
ContextAltDefs :: { [PtContextTermDef] }
     : ContextAltDefs '|' ContextAltDef { $3:$1 }
     | ContextAltDef              { [$1] }

ContextAltDef :: { PtContextTermDef }
     : ContextTermDef    { $1 }
     | rstandard '(' NameDef ')'
         { PtCRstandardv (plGetPosition $1) $3 }
     | rstandard    { PtCRstandard (plGetPosition $1)}
     | cstandard '(' NameDef ')'
         { PtCCstandardv (plGetPosition $1) $3 }
     | cstandard    { PtCCstandard (plGetPosition $1)}
```

Wir überspringen die Definition der Kontextterme (da wir stattdessen die ähnlich aufgebauten Metaterme näher betrachten), die Optionen und die syntaktischen Domänen (da diese ähnlich zu den Kontexten aufgebaut sind) und wenden uns den Regeln zu:

Da man nicht so leicht entscheiden kann, welche Regeln zusammen gehören, lesen wir für jede Regel die Optionen für einen Operator (inkl. Striktheits-Definition – kann auch leer sein) mit ein und fassen die Operatordefinitionen erst bei nachfolgender Verarbeitung richtig zusammen. Wir unterscheiden nun noch zwischen einer Konstruktor-Definition und einer Regel:

```
RedDef  :: { PtRedDef }
     : SwitchDefs1 RuleDef SwitchDefs2
        { (PtRedDef $1 $3 (PtRedRule $2)) }
     | SwitchDefs1 CanonicalDef    { (PtRedDef $1 [] $2) }
```

`SwitchDefs1` beschreibt die Optionen für einen Operator, `SwitchDefs2` den Schalter für die Regel (da wir zukünftig evtl. mehrere Schalter benötigen, sehen wir gleich eine Liste vor). Auch hier sind die Details der Optionen nicht so sehr interessant, allerdings verdient der Umgang mit Zahlen in der Striktheits-Definition noch eine genauere Betrachtung: Wir behandeln Zahlen zunächst wie Namen, prüfen dann aber sofort, ob der String ausschließlich Ziffern enthält und liefern den Wert mit `read` konvertiert zurück oder eine entsprechende Fehler-Information, falls der String nicht nur Ziffern enthält. Zur expliziten Angabe des monadischen Typs müssen wir ein `%` voranstellen, sonst wird die Rückgabe automatisch gekapselt:

```
NumDef    :: { PtNumDef }
     : NameDef {% let (PrsElement numstr pos) = $1 in
                  if (all isDigit numstr) then
                    pmReturn (PrsElement (read numstr) pos)
                  else
                    pmFail
                     (PrsErrorInfo errNumExpectedInt pos)}
```

Eine Konstruktordefinition und eine Auswertungregel sind nun recht ähnlich: Beide enthalten den Operatornamen, die gewünschten Operanden (von Klammern umschlossen), wobei diese bei der Konstruktordefinition nur zur Bestimmung der Stelligkeit dienen, gefolgt von einem Gleichheitszeichen und schließlich entweder dem Schlüsselwort `constructor` oder dem Metaterm, der das Ergebnis der Regel darstellt:

```
CanonicalDef :: { PtRedRuleDef }
     : NameDef ArgsDef '=' constructor { PtCanonical $1 $2 }

RuleDef :: { PtRuleDef }
     : NameDef ArgsDef '=' TermDef { (PtRule $1 $2 $4) }
```

Wir behandeln die Operanden nicht einfach als Liste von Metatermen, stattdessen stellen wir sicher, dass nur ein Operand eine Abstraktion enthalten darf:

```
ArgsDef :: { PtTermsDef }
     : '(' TermsDef ')' { $2 }
     | '(' ')' { [] }

TermsDef :: { PtTermsDef }
     : TermsDef ',' ArgTermDef { $3:$1 }
     | ArgTermDef         { [$1] }

ArgTermDef :: { PtTermDef }
     : VarsDef '.' TermDef { (PtAbstraction $1 $3) }
     | TermDef { $1 }

VarsDef   :: { [PtNameDef] }
     : VarsDef NameDef     { $2:$1 }
     | NameDef           { [$1] }
```

Somit kann ein Metaterm keine äußere Abstraktion enthalten:

```
TermDef   :: { PtTermDef }
     : NameDef ArgsDef { (PtOperator $1 $2) }
     | NameDef '[' TermDef ']' { (PtContext $1 $3) }
     | NameDef '{' TermsDef '}' { (PtMetaApp $1 $3) }
     | NameDef { (PtIdentifier $1) }
```

Namen wollen wir immer zusammen mit ihrer Position im Quelltext speichern, daher lesen wir diese aus den Token mittels einer leicht zu definierenden Funktion `plGetName` heraus:

```
NameDef :: { PtNameDef }
     : name             { (plGetName $1) }
```

Somit haben wir nun die wichtigen Aspekte der Grammatik vollständig beleuchtet. Neben der Grammatik müssen wir auch noch eine Funktion mit vorgegebenem Namen `happyError` definieren, die im Falle eines Parse-Fehlers aufgerufen wird. Dazu können wir in der Happy-Grammatikdatei ein Nachspann verwenden, der mit geschweiften Klammern gekennzeichnet wird, so dass auch hier der Haskell-Code unverändert übernommen wird. Die Fehler-Funktion holt sich dabei mittels (der bei unseren Überlegungen zum „Threaded Lexer“ eingeführten Funktion) `pmGetToken` das Token, das zum Fehler führte und ruft dann mit der entsprechenden Fehler-Information bestehend aus Fehler-Nr. und Position unsere spezielle Funktion `pmFail` auf, die diese Information innerhalb der monadischen Verarbeitung des Parsers kapselt:

```
{
happyError = pmGetToken `pmThen` \(PrsElement t pos) ->
     let pFail n = (pmFail (PrsErrorInfo n pos)) in
          if t == PltLexEOF then
               (pFail errNumEOF)
          else
               (pFail errNumParseError)
}
```

Wie bereits erklärt, müssen aber auch noch alle Listen innerhalb der entsprechenden Datentypen nach dem Parse-Vorgang umgedreht werden (die entsprechenden Funktionen `pReverseInput` und `pReverseTerm` werden einfach entsprechend der Struktur des Datentyps definiert), so dass wir die Parse-Funktionen wie folgt kapseln:

```
pParseRS :: String -> String -> ParseResult PtInputDef
pParseRS filename inputstr =
  let pos = (PrsPosition filename 1 1) in
    do
      prsin <- (pParseRSFunction inputstr pos
                          (PrsElement PltLexEOF pos))
      return (pReverseInput prsin)

pParseTerm :: String -> String -> ParseResult PtTermDef
pParseTerm filename inputstr =
  let pos = (PrsPosition filename 1 1) in
    do
      prst <- (pParseTermFunction inputstr pos
                          (PrsElement PltLexEOF pos))
      return (pReverseTerm prst)
```

Die beiden Funktionen benötigen dabei als Eingabe den Dateinamen, der ja für die Positionsangaben benötigt wird, und den eigentlichen Eingabestring, der also schon vorher aus der Datei heraus gelesen werden muss.

6.2.2.3 Zusammensetzen des Reduktionssystems

Nach dem Parse-Vorgang schließt sich nun das eigentliche Zusammensetzen des Reduktionssystems an, d.h. wir benötigen nun eine Funktion, die die geparste Eingabe vom Typ `PtRedSysDef` zu dem gewünschten Typ `IfrsRedSys` weiterverarbeitet. Da auch während dieses Zusammensetzens Definitions-Fehler abgefangen werden sollen, hat diese Funktion folgenden Typ:

```
brsBuildRedSys :: PtRedSysDef -> ParseResult IfrsRedSys
```

Das Zusammensetzen gliedern wir dabei in fünf Phasen:

1. Zunächst müssen alle Kontexte, Domains und Regeln der Operator-Definitionen zusammengefasst werden,
2. danach werden die Stelligkeiten der Operatoren berechnet,
3. dann werden die Striktheits-Einstellungen verarbeitet,
4. schließlich werden die Regeln zusammengesetzt
5. und zu guter Letzt die Kontexte, Domains und globalen Einstellungen.

Unsere Funktion gestaltet sich somit wie folgt:

```
brsBuildRedSys prs =
  do
    rs0 <- brs1GroupRS prs
    rs1 <- brs2AritiesRS rs0
    rs2 <- brs3StrictRS rs1
    rs3 <- brs4OpRulesRS rs2
    rs  <- brs5BuildRS rs3
    return rs
```

Die einzelnen Phasen gestalten sich aufgrund vieler kleiner Details recht aufwendig, wir konzentrieren uns deshalb im Folgenden auf die elementaren Bestandteile:

Phase 1:

Da wir in unserer Happy-Grammatik aus Abschnitt 6.2.2.2 erlaubt haben, wechselweise Kontexte, Domänen oder Regeln zu definieren, benötigen wir hier eine Funktion, die eine Liste des Typs `PtCDRDef`, der wahlweise eine Kontextdefinition, Domain-Definition oder eine Regel bzw. Konstruktordefinition enthält, umwandelt in drei einzelne Listen, die jeweils alle Kontexte, Domains bzw. Operatordefinitionen enthalten:

```
brs1GroupCDR :: [PtCDRDef] ->
  ParseResult ([PtContextDef], [PtDomainDef], [BrstOpDef1])
```

In den Operatordefinitionen sollen dabei alle Regeln für einen Operator zusammengefasst werden. Es wird erwartet, dass die Regeln für einen Operator auch bereits in der Eingabe hintereinander stehen und nicht durch andere Definitionen unterbrochen werden – ansonsten ist dies ein Fehler! In der Implementierung der Funktion `brs1GroupCDR` lesen wir somit, sobald wir auf die erste Regel eines Operators treffen, die weiteren Regeln die zu diesem Operator passen ein, bis wir auf eine andere, nicht passende Definition treffen. Danach stellen wir sicher, dass in den restlichen Definition keine weitere Regel für diesen Operator definiert wird.

Phase 2:

In dieser Phase werden nun die Metaterme auf den linken Seiten der Regeln untersucht: Aus der Betrachtung, ob die Operanden eine äußere Abstraktion enthalten und falls ja, wie viele Variablen durch diese gebunden werden, kann so leicht die Stelligkeit bestimmt werden. Da wir schon mal dabei sind, die Operanden der Regel näher zu betrachten, bereiten wir auch gleich die Verarbeitung der Striktheits-Information vor, indem wir zusätzlich zur Stelligkeit noch ein Striktheits-Flag zurückliefern, das `True` ist, falls der Metaterm aus einer Anwendung eines Operators auf seine Operanden besteht; wir benutzen dafür ab jetzt den Begriff *Operatorterm*. Die Funktion, die die Metaterm-Definition untersucht, hat somit folgenden Typ:

```
brs2ArityTerm :: PtTermDef -> ParseResult (Bool,OpArityNum)
```

Mittels Pattern-Matching kann man leicht zuordnen, was bei welcher Term-Struktur passieren soll.

Anschließend muss nur noch sichergestellt werden, dass aus allen Regeln für einen Operator die gleichen Stelligkeiten hervorgehen.

Phase 3:

Nachdem wir in Phase 2 bereits die Operanden mit Striktheits-Flags versehen haben, gilt es nun, die Striktheits-Information aus den Operator-Optionen mitzuberücksichtigen. Die wesentliche Funktion ist hier `brs3StrictOpDef`, die eine Operator-Definition aus Phase 2 umwandelt in eine Phase 3 - Definition:

```
brs3StrictOpDef :: BrstOpDef2 ->  ParseResult BrstOpDef3
```

Die Phase 2 - Definition besteht aus dem Operatornamen, der Stelligkeit inkl. Striktheits-Flag, den Operator-Optionen und den Regeln:

```
brs3StrictOpDef (BrstOpDef2 namedef arity opsw rdefs) =
  do
```

Da wir nicht mehr eine Liste von Flags speichern wollen, sondern eine Liste der Positionen, nummerieren wir zunächst die Stelligkeit (mit Striktheits-Flags):

```
    arsd <- return (zip [(1::OpStrictNum)..] arity)
```

Nun lesen wir die benutzerdefinierten Einstellungen aus den Operator-Optionen:

```
    (sd,ex)   <- brs3GetStrictDef opsw True
```

Wie in den Vorüberlegungen beschrieben, kann dabei auch durch das Schlüsselwort `exclusive` festgelegt werden, dass die berechnete Striktheits-Information keine zusätzlichen Positionen enthalten darf. Der Vergleich der benutzerdefinierten und der vorberechneten Striktheits-Information erfolgt mittels der Funktion `brs3AritStrictStrict`:

```
    (sdef,arity') <- brs3AritStrictStrict arsd sd ex
```

Das Ergebnis kann jetzt zurückgeliefert werden, wobei noch die nicht mehr benötigten Striktheits-Angaben aus den Optionen herausgefiltert werden:

```
    return (BrstOpDef3 namedef sdef arity'
              (brs3OpswFilterStrict opsw) rdefs)
```

Die Funktion `brs3AritStrictStrict` kann dabei leicht implementiert werden, wenn die benutzerdefinierten strikten Positionen von `brs3GetStrictDef` auch gleich sortiert zurückgeliefert werden: Es reicht nun, jeweils die Listenköpfe der berechneten Stelligkeit mit Striktheits-Flag, deren Positionen ja durch die vorhergehende Nummerierung markiert sind, und der benutzerdefinierten strikten Positionen zu vergleichen. Falls die Position der berechneten Liste kleiner ist als die nächste benutzerdefinierte Position, muss diese zu den strikten Positionen hinzugefügt werden, falls das Striktheits-Flag `True` ist bzw. falls `exclusive` verwendet wurde, führt dies zu einem Fehler. Die berechnete Liste wird dann solange weiter betrachtet, bis die Position mit der nächsten Benutzerdefinition übereinstimmt. In diesem Fall muss nur überprüft werden, dass die Stelligkeit hier 0 ist, da

sonst keine strikte Position erlaubt ist. Falls die berechnete Liste bereits leer ist, aber immer noch eine benutzerdefinierte Position übrig ist, bedeutet dies, dass diese Position größer ist, als die Anzahl der erlaubten Operanden, so dass dies auch zu einem Fehler führt. So ganz nebenbei wird bei dieser Berechnung noch das Striktheits-Flag aus der Stelligkeit entfernt, so dass die Stelligkeiten und die Striktheits-Informationen nach Abschluss der Phase 3 ihre endgültige Form haben.

Phase 4:

Nun geht es darum, die Regeln zusammenzusetzen, das heißt insbesondere, dass wir für alle Operatornamen unseren Operator-Typ mit Stelligkeit und Striktheits-Information einsetzen, ebenso die sonstigen Namen durch Metavariablen, Variablen oder Domain-Referenzen ersetzen. Dazu müssen wir uns merken, welche Metavariablen auf der linken Seite verwendet wurden; Namen die wir auf der rechten Seite nicht zuordnen können, werden dann zu Variablen. Ebenso müssen wir uns die verwendeten Kontexte und Domains auf der linken Seite merken, da alle auf der rechten Seite verwendeten Kontexte und Domains auf der linken Seite vorkommen müssen. Das Zusammensetzen einer Regel erfolgt anhand der Funktion `brs4BuildRule`, diese erhält das Reduktionssystem aus Phase 3 (um Operator-Einstellungen, Kontext und Domains nachschlagen zu können), die Striktheits-Information des aktuellen Regel-Operators (um Werte-Metavariablen erkennen zu können), die Stelligkeit und natürlich die Regel-Definition, die verarbeitet werden soll:

```
brs4BuildRule :: BrstRedSys3 -> [OpStrictNum] ->
     [OpArityNum] -> BrstRule1 -> ParseResult RuleP

brs4BuildRule s strictdef arity
                    (rsw,r@(PtRule namedef terms term)) =
  do
```

Zuerst werden also die Metaterme der linken Seite zusammen gesetzt, wobei alle verwendeten Metavariablen, Kontexte und Domains gemerkt werden. Die Metaterme werden bei der Übergabe an die Funktion `brs4BuildLHS` mit ihrer Position markiert, damit diese einfach der Reihe nach abgearbeitet werden können:

```
    (terms',mvars,cns,dns)<-(brs4BuildLHS
                              (zip[(1::OpStrictNum)..] terms)
                                 arity strictdef s)
```

Danach wird sichergestellt, dass es bei der Verwendung der Metavariablen keine Widersprüche gibt (denn an dieser Stelle wird noch nicht vorausgesetzt, dass Metavariablen nicht wiederholt verwendet werden, dies geschieht erst bei der Überprüfung der GDSOS-Bedingungen im Analyse-Teil):

```
    ok <- brs4CompareMvars mvars
```

Bei Kontexten und Domains hingegen stellen wir gleich hier sicher, dass diese nicht wiederholt verwendet werden:

```
    ok <- prsListLinear cns errNumRepeatedLeftContext
    ok <- prsListLinear cns errNumRepeatedLeftDomain
```

Nachdem nun die Metavariablen, Kontexte und Domains überprüft wurden, können wir die rechte Seite der Regel zusammensetzen:

```
term' <- brs4BuildRTerm s mvars cns dns (term,0)
```

Die Regel-Einstellungen werden auch noch verarbeitet:

```
rsw' <- (brs4RuleSwitches rsw)
```

Und zum Schluss wird der Operator in die Regel eingesetzt:

```
let op = (OperatorP (prsElement namedef) (prsElPos
            namedef) arity False strictdef) in
  return (RuleP op rsw' terms' term')
```

In der Funktion `brs4BuildLHS` können die Metaterme, wie erwähnt, durch die Markierung mit ihrer Position einfach der Reihe nach bearbeitet werden. Es ist somit nur noch eine Fallunterscheidung über die Termstruktur notwendig:

- Bei einem Identifier bzw. Namen müssen wir, da wir Domain-Referenzen nicht speziell markiert haben, nachschlagen, ob es eine passende Domain-Definition gibt. Falls ja, liefern wir die Domain-Referenz zurück, ansonsten eine Metavariable.

- Bei Abstraktionen müssen wir sicherstellen, dass diese die erlaubten Formen (siehe dazu die entsprechenden Festlegungen in Abschnitt 6.2.2.1) haben, dazu müssen wir die Termstruktur lediglich noch eine Ebene tiefer betrachten.

- Bei einer Verwendung eines Kontextes müssen wir sicher stellen, dass der Kontext auch wirklich definiert wurde, außerdem darf dieser nur eine Metavariable enthalten. Hier sind dann (neben dem zusammengesetzten Metaterm) der Kontextname und die Metavariable zurückzuliefern.

- Besteht der Metaterm aus der Anwendung eines Operators auf seine Operanden, so müssen wir die Operator-Definition nachschlagen und können dann die Operanden rekursiv betrachten.

Die Funktion `brs4BuildRTerm` ist auf ähnliche Weise leicht zu implementieren, hier müssen eben Identifier bzw. Namen in den von `brs4BuildLHS` zurückgelieferten Metavariablen- und Domainnamenslisten nachgeschlagen werden – wie bereits erwähnt werden dabei Namen, die nicht zugeordnet werden können, einfach zu Variablen. Natürlich dürfen auf der rechten Seite aber nun beliebige Abstraktionen und auch Meta-Applikationen benutzt werden, lediglich die Stelligkeit der Operanden muss zu den Operatoren passen und ein Kontext muss bereits auf der linken Seite vorkommen, was ebenfalls leicht durch Nachschlagen in der übergebenen Kontextliste überprüft werden kann.

Wir bereiten nun noch das Zusammensetzen der Kontexte in der nächsten Phase vor, indem wir die Berechnung der Standard-Definitionen für `rstandard` und `cstandard` vorbereiten: Wir speichern nämlich zu jedem Operator, bei dem diese Definitionen nicht durch den Benutzer geändert wurden, entsprechende generierte Definitionen. Diese

speichern wir so, dass sie nicht mehr von einer Benutzer-Definition zu unterscheiden ist, so dass wir beim Zusammenstellen der Kontextalternativen für `rstandard` bzw. `cstandard` nur noch alle Operator-Definitionen durchlaufen und die dortigen Kontext-Definitionen verarbeiten müssen. Damit wir trotzdem noch erkennen können, dass die Definition vom Benutzer geändert wurde, müssen wir ein zusätzliches Flag abspeichern, das bei einer Beeinflussung durch den Benutzer den Wert `True` hat und sonst `False`.

Die automatische Generierung der Standard-Definitionen orientiert sich dabei an Definition 5.2.6. für `rstandard` bzw. Definition 5.1.8. für `cstandard`. Wesentlich ist hier eine Funktion, die aus der Stelligkeit und einer spezifizierten Position, in der rekursiv auf die eigene Kontext-Definition verwiesen werden soll, eine Kontext-Alternative erzeugt, bzw. da der entsprechende Kontextterm immer aus der Anwendung des aktuell verarbeiteten Operators auf seine Operanden besteht, müssen nur die als Operanden erforderlichen Kontextterme berechnet werden. Damit die Stelligkeitsliste dabei einfach durchlaufen werden kann, muss vorher die Stelligkeit für jeden Operator mit ihrer Position versehen worden sein, d.h. die Stelligkeitsliste besteht hier aus Tupeln, die die Stelligkeit und die Position enthalten. Da Positionsangaben den Typ `OpStrictNum` haben, hat die Funktion somit folgenden Typ:

```
brs4BuildCstandardTermArity :: [(OpArityNum,OpStrictNum)] ->
                        OpStrictNum -> [PtContextTermDef]
```

Wir erstellen also die Liste der Operanden für jede Position. Falls wir bereits alle Positionen durchlaufen haben, liefern wir die leere Liste zurück:

```
brs4BuildCstandardTermArity [] x = []
```

Ansonsten wird für eine Stelligkeit *i* und Position *n* ein Operand $v_1 \ldots v_i \,.\, x_n$ erzeugt, es sei denn, *n* entspricht der spezifizierten Position, in der rekursiv auf die eigene Kontext-Definition verwiesen werden soll, in diesem Fall wird anstelle von x_n das Schlüsselwort `recursive` generiert:

```
brs4BuildCstandardTermArity ((a,n):as) x =
  if a==0 then
    c:(brs4BuildCstandardTermArity as x)
  else
    let vars = [(PrsElement ("v" ++ (show v))
                (PrsPosition ":computed automatically:" 1 1))
                    |v<-[1..a]] in
      ((PtCAbstraction vars c):
          (brs4BuildCstandardTermArity as x))
 where
  c=(if x==n then
      (PtCRecursive (PtPosition
          (PrsPosition ":computed automatically:" 1 1)))
     else
      (PtCIdentifier (PrsElement ("x" ++ (show n))
          (PrsPosition ":computed automatically:" 1 1))))
```

Diese Funktion wird nun bei der `rstandard`-Generierung für jede strikte Position aufgerufen, bzw. bei `cstandard` für alle Positionen.

Phase 5:

In dieser Phase werden die Kontext-Definitionen, die Domain-Definitionen und schließlich das Reduktionssystem in der endgültigen Form erstellt. Beim Verarbeiten einer Kontext-Definitionen werden nacheinander die einzelnen Kontextalternativen zusammengesetzt, dann werden noch die Optionen verarbeitet und anschließend kann der neue Kontext-Datentyp zurückgeliefert werden:

```
brs5BuildContext :: BrstRedSys4 ->
  PtContextDef -> ParseResult ContextP

brs5BuildContext sys
     (PtContextDef cpos cname cterms coptions) =
  do
    ctermss' <- mapM (brs5BuildContextTerms sys cname) cterms
    coptions' <- brs5BuildContextOptions coptions
    return (ContextP cname (concat ctermss') coptions')
```

Dabei ist allerdings zu beachten, dass bei Verwendung von Kontextalternativen `rstandard` oder `cstandard` nicht ein einzelner Kontextterm zur Definition hinzugefügt wird, sondern eine Liste von Kontexttermen. Daher erhalten wir also bei der Verarbeitung der Liste der Kontextalternativen eine Liste von Kontextterm-Listen, die dann erst noch konkateniert werden muss, bevor sie in die Kontext-Definition eingesetzt werden kann.

Betrachten wir nun die Funktion `brs5BuildContextTerms`, die eine Kontextalternative verarbeitet: Da bei der Verarbeitung der Definitionen für `rstandard` und `cstandard` (die sich aus den benutzerdefinierten und generierten Standard-Definitionen für jeden Operator zusammensetzen) für das Schlüsselwort `recursive` der aktuelle Kontextname eingesetzt werden soll, muss dieser ebenso übergeben werden wie das gesamte Reduktionssystem (aus Phase 4), um ggf. andere Definitionen nachschlagen zu können. Der Typ der Funktion ist somit:

```
brs5BuildContextTerms :: BrstRedSys4 -> PtNameDef ->
  PtContextTermDef -> ParseResult [ContextTermP]
```

Bei der Implementierung wird eine Fallunterscheidung über die geparste Kontext-alternative vom Typ `PtContextTermDef` gemacht:

- Bei einem Loch wird der Konstruktor `CtEmptyP` zusammen mit der Position der Definition als einziges Listenelement zurückgeliefert.

- Bei einem Identifier bzw. Namen wird überprüft, ob ein Kontext mit diesem Namen definiert wurde. Falls ja, wird eine Referenz auf den Kontext zurückgeliefert, falls nein, ist dies ein Fehler.

- Bei einer Kontext-Benutzung muss sichergestellt werden, dass der Kontext definiert wurde und der in das Loch eingesetzte Kontextterm genau ein Loch enthält.

- Bei einem Operatorterm muss sicher gestellt werden, dass die Operanden genau ein Loch enthalten.

- Bei `rstandard` bzw. `cstandard` müssen die in Phase 4 vorberechneten (bzw. möglicherweise vom Benutzer selbst spezifizierten) Definitionen für jeden Operator betrachtet und entsprechend alle Kontextterme hinzufügt werden.

- Abstraktionen sind auf oberster Ebene nicht erlaubt und führen zu einem Fehler.

Das Zusammensetzen der Kontext-Unterterme übernimmt dabei die Funktion `brs5BuildContextTermEx`, deren Aufbau im wesentlichen `brs4BuildRTerm`, also der Funktion, die die Metaterme auf den rechten Seiten der Regeln zusammensetzt, entspricht. Es fallen dabei lediglich die Metavariablen weg, dafür kommen Löcher und Kontext-Referenzen hinzu. Der wichtigste Unterschied besteht nun darin, dass dabei zusätzlich die verwendeten Löcher und Kontext-Referenzen (da die entsprechenden Kontextdefinitionen ja dann wieder ein Loch enthalten) zurückgeliefert werden, bzw. besser gesagt deren Positionen. Die Funktion hat somit folgenden Typ:

```
brs5BuildContextTermEx :: BrstRedSys4 ->
  (PtContextTermDef,OpArityNum) ->
    ParseResult ([PrsPosition],ContextTermP)
```

Somit kann die Überprüfung, ob ein soeben zusammengesetzter Kontextterm genau ein Loch enthält, ganz leicht erfolgen, indem die zurückgelieferten Loch-Positionen betrachtet werden:

```
brs5SingleHole :: PrsPosition -> [PrsPosition] -> ParseResult
Bool
brs5SingleHole pos hpl =
  case hpl of
    [] -> prsError errNumNoHole pos
    [x] -> return True
    (x:p:xs) -> prsError errNumMultiHole p
```

Als Fehler-Position, falls der Kontextterm kein Loch enthält, wird dabei die Position des Definitionsnamens übergeben.

Betrachten wir nun noch die Verarbeitung der `rstandard`-Definitionen: Wie bereits beschrieben, benötigen wir neben dem Reduktionssystem aus Phase 4 zum Nachschlagen von Definitionen auch den Namen der Definition, in den die Kontextterme eingefügt werden, so dass dieser für das Schlüsselwort `recursive` eingesetzt werden kann. Zurückgeliefert wird eine Liste von Kontexttermen, außerdem liefern wir noch ein Flag zurück, das angibt, ob diese Standarddefinitionen vom Benutzer beeinflusst wurden (`True`) oder nicht (`False`). Dies wird dazu benötigt, um zu erkennen, ob ein

Reduktionskontext, der als einzige Kontextalternative `rstandard` enthält, sich an Definition 5.2.6. hält oder davon abweicht. Unsere Funktion hat somit folgenden Typ:

```
brs5BuildCRstandard :: BrstRedSys4 -> PtNameDef ->
     ParseResult (Bool,[ContextTermP])
```

Nun werden mittels der Funktion `brs5BuildCstandardv` die in Phase 4 bereitgestellten Definitionen für jeden Operator, die mit dem Selektor `brstRstandard4` erhalten werden, verarbeitet:

```
brs5BuildCRstandard sys@(BrstRedSys4 _ _ _ rs) cname =
  do
    res <- (mapM (brs5BuildCstandardv cname sys)
                              (map brstRstandard4 rs))
```

Die Rückgaben von `brs5BuildCstandardv` bestehen ebenfalls aus einem Flag und einer Kontextliste, so dass nun alle Flags (mit einem logischen Oder) und die Listen zusammengefasst werden. Zusätzlich zu den zurückgelieferten Kontexttermen hängen wir außerdem noch den leeren Kontext an:

```
    let (ed,r) = unzip res in
      return ((or ed),
        ((CtEmptyP
          (PtPosition
               (PrsPosition ":computed automatically:" 1 1)))
              : (concat r)))
```

Die Funktion `brs5BuildCstandardv` entspricht dabei der Funktion `brs5BuildContextTerms`, außer dass bei einer Kontextalternative `ignore` eine leere Liste zurückgeliefert wird und eben für `recursive` eine Referenz auf den übergebenen Kontextnamen. Für die Berechnung der Kontextterme für `cstandard` muss man in der Funktion `brs5BuildCRstandard` nur den Selektor `brstRstandard4` durch `brstCstandard4` austauschen.

Das Zusammensetzen der Domains gestaltet sich analog zu den Kontexten, nur einfacher, da hier keine Löcher und Standard-Definitionen mehr betrachtet werden müssen. Bei den globalen Optionen werden neben den Schaltern auch der Reduktions- und der Programmkontext verarbeitet. Falls diese nicht definiert wurden, wird automatisch eine Definition `reductioncontext = rstandard` bzw. `programcontext = cstandard` verwendet. Die Verarbeitung der Definitionen erfolgt dann mit den üblichen die Kontexte zusammensetzenden Funktionen. Beim Verarbeiten der restlichen Optionen müssen nur die Konstruktoren richtig zugeordnet werden. Zum Schluss folgen noch ein paar einfache Angleichungen an den Reduktionssystem-Datentyp und endlich kann das fertige, eingelesene Reduktionssystem zurückgeliefert werden.

6.2.2.4 Aufruf der Ladeprozedur

Als Einstieg in unsere Ladeprozedur dient nun eine Funktion `loadRS`. Diese bekommt den Dateinamen der gewünschten Eingabedatei übergeben und liefert das eingelesene

Reduktionssystem zurück. Da wir uns beim Einlesen der Eingabedatei der Methoden der IO-Monade bedienen, hat diese Funktion also folgenden Typ:

```
loadRS :: FilePath -> IO IfrsRedSys
```

Die Funktion liest mittels der IO-Methode `readFile` die Eingabedatei und übergibt die Eingabe an unsere Parser-Funktion `pParseRS` aus Abschnitt 6.2.2.2. Beim Parse-Ergebnis wird dann geprüft, ob aufgrund der Verwendung des `include`-Befehls in der Eingabedatei weitere Dateien eingelesen werden müssen. Falls ja, werden diese Dateien ebenfalls geparst und die Ergebnisse (durch Konkatenation der Listen des Typs `PtCDRDef`) kombiniert. Das Gesamtergebnis wird dann der in Abschnitt 6.2.2.3 definierten Funktion `brsBuildRedSys` übergeben. Anschließend müssen wir nur noch das Reduktionssystem aus dem Fehlertyp `ParseResult` extrahieren und können dann das eingelesene Reduktionsystem zurückliefern.

6.2.3 Überprüfung der GDSOS-Bedingungen

Im Analyse-Teil wollen wir nun überprüfen, ob die 5 Bedingungen aus Definition 5.2.5. erfüllt sind, da wir dann sofort die in Abschnitt 5.2.3 beschriebenen Eigenschaften folgern können. Dabei muss allerdings noch zusätzlich sichergestellt werden, dass der Reduktionskontext der Standarddefinition entspricht, also nicht vom Benutzer verändert wurde. Die Bedingung 5, dass jede Metavariable auf der rechten Seite einer Regel auch auf der linken Seite vorkommt, ist bereits durch unsere Einlese-Routine gesichert, da ein unbekannter Name auf der rechten Seite immer zu einer (normalen) Variable wird. Zu überprüfen sind somit noch die Bedingungen 1-4:

Bedingung 1: Metaterme auf der linken Seite einer Regel sind einfache Metawerte, falls sie an strikten Positionen stehen, sonst (Nicht-Werte-)Metavariablen.

Wir benötigen also eine Funktion `mtIsSimpleMval`, die überprüft ob, ein Metaterm einen einfachen Metawert darstellt, und eine Funktion `mtIsMvar`, die überprüft ob ein Metaterm aus einer Metavariablen, ggf. in einer Darstellung der Form $x_1...x_n$. $X\{x_1...x_n\}$, besteht, wobei das Value-Flag `False` sein muss, da es sich sonst um einen einfachen Metawert handelt. Betrachten wir zunächst `mtIsMvar`: Da wir, wie in Abschnitt 6.2.1 erwähnt, eine Multiparameter-Klasse für Metaterme benutzen, bekommt diese Funktion neben dem Metaterm noch eine Umgebung als Argument, zurückgeliefert wird `True`, falls der Metaterm aus einer Metavariablen besteht und `False`, falls er eine andere Struktur hat.

```
mtIsMvar :: MetaTerm mt env v mv op cn dn =>
                    mt -> env -> Bool
```

Eine Metavariable kann nun leicht durch die Methode `mtIsMetaVar` der Klasse `MetaTerm` erkannt werden, mit `mtGetMetaVar` kann diese selektiert werden und mittels der `MetaVar`-Methode `mvarIsValue` kann das Value-Flag der Metavariable geprüft werden:

```
mtIsMvar mt env
  | mtIsMetaVar mt env =
          not (mvarIsValue (mtGetMetaVar mt env) env)
```

Bei Metavariablen der Stelligkeit > 0 ist dies jedoch bei einer Darstellung in der Form $x_1...x_n$. $X\{x_1...x_n\}$ etwas schwieriger, da die Struktur nun wie folgt ist: Eine Abstraktion, die eine Meta-Applikation enthält, deren Unterterme wieder die abstrahierten Variablen sind:

```
  | mtIsAbstraction mt env =
    let term = mtGetAbsTerm mt env in
      if mtIsMetaApp term env then
        let terms = (mtGetMetaAppTerms term env) in
          if all (\x -> mtIsVariable x env) terms then
            ((map (\x -> varName x env)
             (mtGetAbsVars mt env)) ==
              (map (\x -> varName (mtGetVariable x env) env)
                    terms))
          else
            False
      else
        False
```

Bei anderen Termstrukturen kann es sich nicht um eine Metavariable handeln:

```
  | otherwise = False
```

Die Funktion `mtIsSimpleMval` bekommt ebenfalls einen Metaterm samt Umgebung übergeben und liefert `True` zurück, falls der Metaterm ein einfacher Metaterm ist, sonst `False`:

```
mtIsSimpleMval :: MetaTerm mt env v mv op cn dn =>
                          mt -> env -> Bool
```

Gemäß Definition 5.2.4. ist ein einfacher Metawert eine Werte-Metavariable oder eine Anwendung eines kanonischen Operators auf seine Operanden, die an allen strikten Positionen einfache Metawerte sind und an den restlichen Positionen (Nicht-Werte-) Metavariablen. Die Funktion `mtIsSimpleMval` kann somit auch leicht über eine Fallunterscheidung implementiert werden:

```
mtIsSimpleMval mt env

  | mtIsMetaVar mt env =
          (mvarIsValue (mtGetMetaVar mt env) env)

  | mtIsOpTerm mt env =
    let op = mtGetOp mt env in
      -- Operator kanonisch?
      if opCanonical op env then
        let terms = mtGetOpTerms mt env in
```

```
        mtIsSimpleMval1 terms env (opStrict op env) 1
      else
        False

  | otherwise = False
```

Dabei wird eine Hilfsfunktion `mtIsSimpleMval1` benutzt, um die Operanden zu überprüfen, d.h. die Funktion bekommt die Liste der Operanden übergeben, die strikten Positionen, außerdem die Startposition 1, und prüft nun der Reihe nach die Operanden, wobei bei rekursiven Aufrufen die übergebene Position um Eins erhöht wird. So kann leicht überprüft werden, ob die aktuelle Position strikt ist und somit ein einfacher Metawert oder eine Metavariable erforderlich ist. Falls keine Operanden mehr übrig sind, so ist die Bedingung erfüllt.

```
mtIsSimpleMval1 [] _ _ _ = True

mtIsSimpleMval1 (x:xs) env strict n =
  -- strikte Position?
  if (elem n strict) then
    -- einfacher Metawert?
    if (mtIsSimpleMval x env) then
     (mtIsSimpleMval1 xs env strict (n+1))
    else
     False
  else
    -- Metavariable?
    if (mtIsMvar x env) then
     (mtIsSimpleMval1 xs env strict (n+1))
    else
     False
```

Im Interpreter geben wir, wenn Bedingung 1 nicht erfüllt ist, eine Warnung aus, führen dann aber die Auswertung durch (falls keine anderen Gründen dagegen sprechen), da wir u.a. auch im Call-by-Need-Kalkül, welches gegen diese Bedingung verstößt, auswerten möchten.

Bedingung 2: Die linke Seite einer Regel ist linear.

Es muss also überprüft werden, dass keine Metavariable wiederholt auf der linken Seite verwendet wird. Dazu benötigen wir eine Funktion `mtMvarsList`, die die Liste aller Metavariablen, die in einer Liste von Metatermen enthalten sind, zurückliefert:

```
mtMvarsList :: MetaTerm mt env v mv op cn dn =>
                    [mt] -> env -> [mv]
```

Diese Funktion kann leicht über die Struktur der Metaterme und mittels der Selektor-Methoden der Metaterm-Klasse implementiert werden. Anschließend können dann die Namen der Metavariablen überprüft werden und sichergestellt werden, dass kein Name doppelt verwendet wird, indem die Metavariablenliste durchlaufen wird und für jedes

Element der Liste geprüft wird, dass kein Element aus der Restliste den gleichen Namen hat.

Falls Bedingung 2 nicht erfüllt ist, bricht der Interpreter die Auswertung mit einer entsprechenden Fehlermeldung ab, da sonst die Regelsuche zu schwierig würde.

Bedingung 3: Die Regeln sind nicht überlappend.

Wenn wir bereits Bedingung 2 sichergestellt haben, reicht es zu überprüfen, dass alle Regeln für einen Operator jeweils paarweise an einer beliebigen Position unterschiedliche Operatoren enthalten, und zwar, da Metavariablen für beliebige Terme stehen können, in genau dem gleichen Teil in der Struktur eines Unterterms. Wir haben allerdings unterschiedliche Matching-Einstellungen für die Regeln, daher müssen wir diese auch berücksichtigen: Regeln mit dem Schalter `<eval_alt>` dürfen sich überlappen, ohne die Bedingung 3 zu verletzen, da diese Einstellung ja besagt, dass beim Suchen der Regel die erste passende Alternative genommen wird (genauso, wie dies beim Haskell-Pattern Matching der Fall ist). Eine deterministische Auswertung ist somit gewährleistet, d.h. man könnte die Reduktionsregeln auch abändern in unendlich viele, nicht überlappende Regeln, die ein äquivalentes Reduktionsverhalten haben. Zu überprüfen ist somit zunächst einmal, ob sich eine Regel mit dem Schalter `<eval_excl>` mit einer anderen Regel überlappt. Ist dies der Fall, so brechen wir die Auswertung mit einem Fehler ab. Danach prüfen wir, ob nichtdeterministische Regeln mittels des Schalters `<eval_nd>` definiert wurden, falls ja, geben wir im Interpreter eine Warnung aus und führen dann die Reduktion solange durch, bis mehrere Regeln für die weitere Reduktion in Frage kommen.

Betrachten wir nun die Funktion, die zwei Regeln für einen Operator vergleicht; genau genommen reicht es, die Metaterme der beiden linken Seiten zu betrachten, wenn wir im Vorhinein schon sichergestellt haben, dass es sich um Regeln für denselben Operator handelt (ansonsten können die Regeln ja auch nicht überlappen):

```
rulePrsValidThree2 :: MetaTerm mt env v mv op cn dn =>
                [mt] -> [mt] -> env -> Bool
```

Falls keine weiteren Metaterme übrig sind, kann kein Unterschied festgestellt werden:

```
rulePrsValidThree2 [] [] _ = False
```

Wir gehen davon aus, dass die Regeln beim Laden entsprechend Abschnitt 6.2.2.3. auf ihre Form geprüft wurden, so dass Abstraktionen auf der linken Seite nur die Formen $x_1...x_n.X\{x_1,...,x_n\}$ oder $x.C[x]$ oder $x_1...x_n.D$ haben dürfen, wobei X eine Metavariable, C ein Kontext und D eine Domain ist. Außerdem darf ein Metaterm, der eine Kontextbenutzung enthält, aber keine Abstraktion ist, ausschließlich die Form $C[X]$ haben. Da wir es uns in diesem Zusammenhang ersparen wollen, Kontexte und Domains näher zu untersuchen, gehen wir davon aus, dass diese stets so definiert sind, dass ein beliebiger Term darin enthalten sein kann, d.h. wir gehen im Zweifelsfall davon aus, dass sich die Regeln überlappen können. Somit kann also bei einem Vergleich einer Metavariablen, Kontextbenutzung oder Domain mit einem beliebigen Metaterm kein Unterschied festgestellt werden, ebenso bei Abstraktionen, da diese als Unterterme nur eine Metavariable (als Meta-Applikation mit den Bindungsvariablen), eine Kontextbenutzung

mit der Bindungsvariablen oder eine Domäne haben dürfen. Da Meta-Applikationen ohne Abstraktionen auf der linken Seite nicht erlaubt sind, reicht es somit zu überprüfen, ob die jeweiligen nächsten Metaterme beide Operatorterme sind, und falls ja, die Operatoren und die Operanden zu prüfen. Findet sich hier ein Unterschied, so können wir `True` zurückliefern, ansonsten müssen wir rekursiv weiter bei der nächsten Position nach einem Unterschied suchen:

```
rulePrsValidThree2 (r1:rs1) (r2:rs2) env

  | (mtIsOpTerm r1 env) && (mtIsOpTerm r2 env) =
    let op1 = mtGetOp r1 env
        op2 = mtGetOp r2 env
        ots1 = mtGetOpTerms r1 env
        ots2 = mtGetOpTerms r2 env
          in
      if opEq op1 env op2 env then
        if rulePrsValidThree2 ots1 ots2 env then
          True
        else
          rulePrsValidThree2 rs1 rs2 env
      else
        True

  | otherwise =
    rulePrsValidThree2 rs1 rs2 env
```

Bedingung 4: Die rechte Seite einer Regel enthält keine freien Variablen.

Wir erstellen hier eine Funktion `mtFreeVars1`, die die Liste aller freien Variablen, die in einem Metaterm enthalten sind, zurückliefert. Diese kann, analog zu der Funktion `mtMvarsList`, die die Liste aller enthaltenen Metavariablen berechnet, leicht über die Struktur der Terme erstellt werden. Wir müssen uns dabei lediglich, wenn wir auf eine Abstraktion treffen, die Bindungsvariablen merken, bzw. diese bei einem rekursiven Aufruf mit übergeben (beim ersten Aufruf übergeben wir einfach die leere Liste) um dann zwischen gebundenen und freien Variablen unterscheiden zu können. Die Funktion hat somit folgenden Typ:

```
mtFreeVars1 :: (MetaTerm mt env v mv op cn dn, Eq v) =>
                    mt -> env -> [v] -> [v]
```

Wenn wir bei der Berechnung auf eine Variable *v* treffen, so schauen wir nach, ob diese in der Liste der gebundenen Variablen enthalten ist, falls nein liefern wir `[`*v*`]` zurück und sonst `[]`. Die Ergebnisse aus allen Untertermen müssen dann nur noch durch `++` verknüpft werden.

Zur Überprüfung der Bedingung 4 reicht es nun, zu untersuchen, ob die Liste der freien Variablen des Metaterms auf der rechten Seite der Regel leer ist: falls nein, brechen wir die Auswertung mit einer entsprechenden Fehlermeldung ab.

Prüfung des Reduktionskontextes:

Wir beschränken uns hier darauf, eine Warnung auszugeben, falls der Reduktionskontext durch den Benutzer geändert wurde; dazu müssen wir lediglich nachschauen, ob das Flag, das wir beim Laden zusammen mit dem Reduktionskontext gespeichert haben, auf `True` (Änderung durch den Benutzer) oder auf `False` steht (d.h. keine Änderung durch den Benutzer, der Reduktionskontext wurde vollständig automatisch gemäß Definition 5.2.6. berechnet).

Aufruf der Analyse:

Wir benötigen nun noch eine Funktion `analyse`, welche die eben beschriebenen Überprüfungen nacheinander durchführt und einen Fehler oder eine Warnung ausgibt, falls die gewünschten Bedingungen nicht gelten. Da wir für die Ausgabe der Warnungen die IO-Methode `putStrLn` verwenden und sonst keine weiteren Rückgaben notwendig sind, hat unsere Funktion folgenden Typ:

```
analyse :: IfrsRedSys -> IO ()
```

In der Implementierung der Funktion müssen wir dann die Operatordefinitionen aus dem Reduktionssystem lesen, aus diesen wiederum die Regeln und die Metaterme der Regeln und diese schließlich auf beschriebene Art und Weise prüfen.

6.2.4 Der Interpreter

Wie bereits in den Vorüberlegungen beschrieben, wollen wir den Interpreter mit Hilfe einer Template Instantiation Machine realisieren, prinzipiell könnte aber auch z.B. eine G-Machine (siehe [Sch01, Kapitel 2] oder [PJL91, Kapitel 3]) oder jede andere abstrakte Maschine, die für die Compilierung verzögert auswertender funktionaler Sprachen entworfen wurde, verwendet werden. Wir entscheiden uns hier für die Template Instantiation Machine, da diese am einfachsten zu implementieren ist.

Die Template Instantiation Machine arbeitet dabei über einem Zustand, in dem u.a. der auszuwertende Term in einer Graph-Darstellung gespeichert ist. Für jeden Reduktionsschritt wird der entsprechende Term dann durch sein Reduktionsergebnis ersetzt, d.h. der Zustand ändert sich bei jedem Reduktionsschritt.

Die grundlegenden Funktionen, die für die Implementierung des Interpreters benötigt werden, sind somit

- eine Funktion `compile`, die unser eingelesenes Reduktionssystem übergeben bekommt und daraus den initialen Zustand der Template Instantiation Machine berechnet,

- und eine Funktion `eval`, die die Auswertung durchführt. Da wir an den einzelnen Reduktionsschritten interessiert sind, liefern wir die Liste der Zustände für jeden Reduktionsschritt zurück, aus den Zuständen können wir dann die entsprechenden Terme extrahieren.

Die Auswertung erfordert dabei folgende Schritte:

- den nächsten Redex finden, dazu muss der Reduktionskontext betrachtet werden,
- diesen reduzieren, d.h. die passende Regel finden und anwenden,
- und schließlich den Redex mit dem Ergebnis überschreiben.

Die Auswertung endet, wenn der Term eine kanonische Form angenommen hat (d.h. ein Konstruktor-Term, bei dem alle strikten Argumente Werte sind) oder in einer `value`-Domain enthalten ist.

Betrachten wir zunächst den Zustand der Template Instantiation Machine genauer:

Der wichtigste Bestandteil ist der *Heap*, der die Terme in ihrer Graphstruktur speichert. Die Heap-Knoten sind dabei an festen Adressen gespeichert, so dass die Anwendung eines Operators auf seine Argumente durch einen Applikations-Knoten dargestellt werden kann, der den Operatornamen, einen Verweis auf die Operatordefinition (inkl. Reduktionsregeln) und die Adressen der Operanden enthält. Im Gegensatz dazu sind in [PJL91] die Applikations-Knoten auf Currying ausgelegt und enthalten genau zwei Argumente, wobei das erste keine Superkombinator-Definition sein muss, sondern auch eine (ungesättigte) Funktionsanwendung sein kann. Es wird dann eine spezielle *Unwind*-Operation benötigt, die den Graphen durchläuft, um die äußerste Funktionsanwendung zu finden.

Da unsere Termstruktur mit festen Stelligkeiten jedoch kein Currying erlaubt und wir daher alle benötigten Verweise direkt in dem Applikations-Knoten speichern, können wir auf eine *Unwind*-Operation verzichten und somit auch auf den *Stack*, der benötigt wird, um sich bei der Unwind-Operation die Rücksprung-Adressen zu merken. Der *Dump*, der zum Sichern von Stacks dient, wenn Zwischenauswertungen nötig werden, kann nun durch einen einfachen Stack ersetzt werden. Allerdings ist auch dieser nicht nötig, wenn wir unsere Suche nach dem nächsten Redex anhand des Reduktionskontextes stets von der Wurzel (d.h. der Adresse des Hauptterms) aus beginnen. Da bei Reduktionssystemen, die sich an die GDSOS-Bedingungen halten, die Suche nach dem nächsten Redex durch eine Einschränkung auf die aktuell betrachteten Unterterme (deren Adressen dann im Stack gespeichert werden) stark beschleunigt werden kann (da es ausreicht die strikten Positionen zu betrachten, d.h. ein Abgleich mit dem Reduktionskontext ist nicht mehr nötig), planen wir den Stack jedoch mit ein.

Neben dem Heap und dem Stack benötigen wir aber auch noch sämtliche Informationen des Reduktionssystems, also die Operatordefinitionen, Kontexte, Domains und die globalen Einstellungen. Außerdem sehen wir einen Statistik-Eintrag vor, in dem Informationen über die Reduktion gespeichert werden können, z.B. die Anzahl der bisher benötigten Reduktionsschritte.

Wir verwenden somit für den Zustand der Template Instantiation Machine folgenden Datentyp:

```
data TiState = TiState {
  tiStack :: TiStack,
  tiHeap :: TiHeap,
  tiStats :: TiStats,
  tiGlobal :: IfrsGlobal,
```

```
  tiContexts :: IfrsContexts,
  tiDomains :: IfrsDomains,
  tiOpDefs :: TiOpDefs
} deriving Show
```

Den Zustand verwenden wir nun bei den in Abschnitt 6.2.1 beschriebenen Klassen als Umgebung. Da wir dabei sämtliche Informationen des Reduktionssystems aus dem Zustand entnehmen können, können wir den leeren Typ `()` zusammen mit dem Zustand als Reduktionssystem benutzen:

```
instance
 (RsGlobal IfrsGlobal TiState (...),

  Contexts IfrsContexts TiState (...),

  Domains IfrsDomains TiState (...),

  OpDefs TiOpDefs TiState IfrsOpDef (...) ) =>

   RedSys () TiState IfrsGlobal IfrsContexts IfrsContext
    IfrsDomains IfrsDomain TiOpDefs IfrsOpDef
    IfrsContextName IfrsContextTerms IfrsContextTerm
    IfrsContextVariable IfrsDomainName IfrsDomainTerms
    IfrsDomainTerm IfrsDomainVariable IfrsRules IfrsRule
    IfrsMetaTerm IfrsOperator IfrsMetaVariable IfrsVariable

     where

      rsGlobal _ env = (tiGlobal env)
      rsContexts _ env = (tiContexts env)
      rsDomains _ env = (tiDomains env)
      rsOpDefs _ env = (tiOpDefs env)
```

Analog zu den Superkombinator-Definitionen in [PJL91], speichern wir die Operatordefinitionen als Heap-Knoten. In einer Assoziationsliste speichern wir dann zu jedem Operatornamen die passende Adresse:

```
type TiOpDefs = Assoc String HeapAddress
```

Den Typ `TiOpDefs` können wir dann wie folgt zu einer Instanz von `OpDefs` machen:

```
instance
  (OpDef IfrsOpDef TiState (...) ) =>

    OpDefs TiOpDefs TiState (...) where

      odList ods env =
          (map (tiGetOpDef (tiHeap env)) (assocRange ods))
```

```
odLookUp n ods env =
  do
    (n,a) <- assocLookElemFst ods n
    return (tiGetOpDef (tiHeap env) a)
```

Bei der Speicherung der Operatordefinitionen in den Heap-Knoten empfiehlt es sich, wie wir gleich sehen werden, Operatoren, die keine Argumente haben, speziell zu behandeln. Daher gibt es also zwei Sorten von Heap-Knoten, die Operatordefinitionen speichern:

```
tiGetOpDef :: TiHeap -> HeapAddress -> IfrsOpDef
tiGetOpDef h a =
  case (heapLookup h a) of
    TiNRule _ od -> od
    TiNCAF _ od _ -> od
    _ -> error "tiGetOpDef: Unexpected heap-node!"
```

Wenden wir uns einer genaueren Betrachtung der Heap-Knoten zu. Wie beschrieben stellen wir die Anwendung eines Operators auf seine Argumente durch einen Applikations-Knoten dar, dieser hat folgende Form:

`TiNAp` *Name Operatordefinition Operanden*

Bei den Knoten, die die Operatordefinitionen enthalten, unterschieden wir, wie bereits angedeutet, zwischen Operatoren, die Argumente erfordern und Operatoren, die keine Operanden haben. Ein nichtkanonischer Operator ohne Operanden stellt nämlich für sich alleine einen Redex dar, falls wir diesen einmal ausgewertet haben, würden wir uns bei einer wiederholten Verwendung gerne eine wiederholte Berechnung sparen und uns stattdessen gleich das Ergebnis merken. Üblicherweise werden solche Operatoren ohne Operanden als *constant application forms* (kurz CAFs) bezeichnet. Während wir also bei Operatoren, die Argumente erfordern, nur den Namen und die Operatordefinition (inkl. Reduktionsregeln) speichern, d.h. der Heap-Knoten hat folgende Form:

`TiNRule` *Name Operatordefinition*,

enthält der Knoten für eine CAF zusätzlich noch die Möglichkeit auf die instanziierte rechte Seite einer entsprechenden Auswertungsregel zu verweisen:

`TiNCAF` *Name Operatordefinition Verweismöglichkeit*

Für die Verweismöglichkeit verwenden wir den Datentyp `Maybe`, der dann `Nothing` enthält, falls der Operator noch nicht reduziert wurde, oder die Heap-Adresse des Reduktionsergebnisses (mit `Just` gekapselt).

Da unsere Operanden auch Abstraktionen sein können (im Gegensatz zu [PJL91], bei der ggf. der Compilierung ein Lambda-Lifter vorgeschaltet wird), benötigen wir auch spezielle Heap-Knoten für Abstraktionen und Variablen.

Der Abstraktionsknoten enthält dabei die Verweise auf die gebundenen Variablen und den abstrahierten Term:

`TiNAbs` *Variablen Term*

Ein Variablenknoten enthält einfach den Variablennamen:

`TiNVar` *Name*

Zum Schluss benötigen wir noch einen Indirektionsknoten. Dieser wird benötigt, falls ein Operator sein Argument unausgewertet zurückliefert (siehe [PJL91, Abschnitt 2.1.5]). Da wir den Argumentterm beim Überschreiben des Redexes nicht kopieren wollen, erzeugen wir in diesem Fall einen Indirektionsknoten, der einfach nur einen Verweis auf den Argumentterm enthält, d.h. der Indirektionsknoten enthält nichts weiter als eine Heap-Adresse:

`TiNInd` *Adresse*

Wir können einen Heap-Knoten somit durch folgenden Datentypen darstellen:

```
data TiNode = TiNAp TiName HeapAddress [HeapAddress]
     | TiNRule TiName IfrsOpDef
     | TiNCAF TiName IfrsOpDef (Maybe HeapAddress)
     | TiNAbs [HeapAddress] HeapAddress
     | TiNInd HeapAddress
     | TiNVar TiName
     deriving (Eq, Show)

type TiName = String
```

Heap und Stack in unserem Zustand haben nun folgende Typen:

```
type TiHeap = Heap TiNode

type TiStack = Stack HeapAddress
```

Für eine kurze Beschreibung der Heap- und Stack-Datenstrukturen und Funktionen siehe Anhang. Bei dem Statistik-Eintrag speichern wir zunächst nur die Anzahl der Schritte, es reicht somit erst einmal der Typ `Int` aus:

```
type TiStats = Int
```

Bevor wir uns den Funktionen der Template Instantiation Machine zuwenden, betrachten wir noch kurz, wie wir Heap-Adressen als Terme verwenden können:

```
instance Term HeapAddress TiState TiName IfrsOperator where
  trmIsVariable a st = tiTermIsVariable st a
  trmIsAbstraction a st = tiTermIsAbstraction st a
  trmIsOpTerm a st = tiTermIsOpTerm st a
  trmGetVariable a st = tiTermGetVariable st a
  trmGetAbsVars a st = tiTermGetAbsVars st a
  trmGetAbsTerm a st = tiTermGetAbsTermAddr st a
  trmGetOp a st = tiTermGetOp st a
```

```
  trmGetOpTerms a st = tiTermGetOpTerms st a
```

Die Zugriffsfunktionen `tiTerm`... sind dabei leicht über die Struktur unserer Heap-Knoten zu implementieren. Betrachten wir dazu beispielhaft `tiTermIsVariable`:

Die Funktion bekommt den aktuellen Zustand und die gewünschte Heap-Adresse übergeben und liefert `True` zurück, falls der Term ausschließlich eine Variable enthält, sonst `False`.

```
tiTermIsVariable :: TiState -> HeapAddress -> Bool
```

Wir betrachten dazu einfach den Heap-Knoten an der übergebenen Adresse und folgen ggf. Indirektionen. Falls wir einen `TiNVar`-Knoten vorfinden liefern wir `True` zurück, bei anderen Knoten, als Variablen und Indirektionen, liefern wir `False` zurück.

```
tiTermIsVariable st a =
  case heapLookup (tiHeap st) a of
    TiNVar n -> True
    TiNInd addr -> tiTermIsVariable st addr
    _ -> False
```

6.2.4.1 Der Compiler

Wir implementieren zunächst eine Funktion `tiCompile`, die das eingelesene Reduktionssystem vom Typ `IfrsRedSys` umwandelt in einen Zustand der Template Instantiation Machine, der dem initialen Zustand entspricht, außer dass der `main`-Term noch nicht instanziiert wurde und der Stack leer ist. Der `main`-Term wird dann in der Funktion `tiStart` instanziiert, bzw. da `main()` eine CAF ist, reicht es, die Adresse von `main` in der `TiOpDefs`-Assoziationsliste zu finden und auf den Stack zu legen. Da wir allerdings auch optional ohne Sharing (und ohne Merken der CAF-Reduktionsergebnisse) auswerten wollen, müssen wir dann bei einer globalen Einstellung

```
sharing = off
```

einen neuen Applikations-Knoten

```
TiNAp "main" address_of_main []
```

erzeugen, wobei `address_of_main` die Adresse der Operatordefinition von `main` ist, und die Adresse dieses neuen Knotens in den Stack legen.

Betrachten wir nun die Funktion `tiCompile`:

```
tiCompile :: IfrsRedSys -> TiState
tiCompile rs =
  let ts = (TiState stackInit initial_heap
              tiInitStat global contexts domains opdefs)
      global = rsGlobal rs ()
      contexts = rsContexts rs ()
```

```
      domains = rsDomains rs ()
      sc_defs = odList (rsOpDefs rs ts) ts
      (initial_heap,opdefs) = tiBuildInitialHeap sc_defs ts
          in
    ts
```

Etwas verwirrend mag hier erscheinen, dass die Funktion `tiBuildInitialHeap`, die einen Bestandteil des Zustands errechnen soll, bereits den Zustand übergeben bekommt. Da diese Funktion aber nicht auf den Heap und die Operatordefinitionen des übergebenen Zustands zugreift, ist dies aufgrund der verzögerten Auswertung problemlos möglich. Das Erstellen des Heaps ist hierbei auch der interessante Vorgang, denn die globalen Einstellungen, die Kontexte und Domains werden einfach unverändert in den neuen Zustand übernommen, die Funktion `stackInit` liefert einen leeren Stack zurück und `tiInitStat` setzt den Zähler der Statistik-Einstellung auf `0`.

Die Funktion `tiBuildInitialHeap` erstellt nun für alle Operatoren `TiNRule`- oder `TiNCAF`-Knoten und merkt sich dabei auch gleich deren Adressen und liefert somit neben dem neuen Heap auch die Assoziationsliste der Operatordefinitionen zurück:

```
tiBuildInitialHeap :: [IfrsOpDef] -> TiState ->
                                        (TiHeap, TiOpDefs)
```

In der Implementierung behandeln wir die Elemente der Eingabeliste nacheinander anhand einer Funktion `tiAllocateSc`. Da diese jedoch den Heap verändert, müssen wir aufpassen, dass für jedes neue Element auch der neue Heap übergeben wird. Dies leistet eine besondere Mapping-Funktion `mapAccumL` (aus der Haskell List-Library), die wir im Anschluss an die Implementierung der Funktion `tiBuildInitialHeap` betrachten. Die Implementierung gestaltet sich wie folgt:

```
tiBuildInitialHeap opdefs env =
  let (h,gl) = mapAccumL alloc heapInit opdefs
      alloc = (\h od -> tiAllocateSc h od env)    in
    (h, (Assoc gl))
```

Wie man sieht, wird jetzt einfach die Funktion `mapAccumL` mit den Argumenten `alloc`, `heapInit` und `opdefs` aufgerufen, wobei die lokal definierte Funktion `alloc` einfach `tiAllocateSc` mit der übergebenen Umgebung aufruft, die Funktion `heapInit` einen neuen Heap (mit unbenutzten Speicherplätzen) erstellt und `opdefs` die übergebenen Operatordefinitionen aus dem eingelesenen Reduktionssystem sind. Zurückgeliefert wird dann der neue Heap `h` und eine Liste `gl`, die Tupel bestehend aus einem Operatornamen und der zugehörigen Adresse des neuen `TiNRule`- oder `TiNCAF`-Knotens enthält. Um die Liste `gl` als Assoziationsliste verwenden zu können, wird diese bei der Rückgabe mit dem Konstruktor `Assoc` gekapselt.

Betrachten wir nun die Definition der Funktion `mapAccumL` aus dem im Haskell-Sprachumfang enthaltenen Modul `List`:

```
mapAccumL :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
```

```
mapAccumL f s []        =  (s, [])
mapAccumL f s (x:xs)    =  (s'',y:ys)
                           where (s', y ) = f s x
                                 (s'',ys) = mapAccumL f s' xs
```

Als Funktionsargumente werden eine verarbeitende Funktion `f` vom Typ `(a -> b -> (a, c))`, ein sogenannter Akkumulator `s` vom Typ `a` (der durch die übergebene Funktion verändert wird) und eine Eingabeliste vom Typ `[b]` erwartet. Die Funktion `f` wird auf jedes Element der Eingabe zusammen mit dem Akkumulator angewendet, wobei jeweils ein Tupel zurückgeliefert wird, das den veränderten Akkumulator und ein Ausgabeelement vom Typ `c` enthält. Das nächste Eingabeelement wird zusammen mit dem veränderten Akkumulator an die verarbeitende Funktion übergeben und alle Rückgabe-Elemente werden zu einer Liste zusammengefasst. Im Zusammenhang mit dem Compiler ist der Akkumulator der Heap, die Eingabeliste enthält die Operatordefinitionen und die Ausgabeliste die Zuordnung der Adressen der neuen Knoten zu den Operatornamen.

Nun fehlt nur noch die verarbeitende Funktion `tiAllocateSc`, die für eine Operatordefinition den passenden Heap-Knoten erzeugt. Falls der Operator keine Argumente hat, d.h. die Stelligkeit in der Listendarstellung ist gleich der leeren Liste, so wird ein Knoten (`TiNCAF` *Operatorname Operatordefinition* `Nothing`) erzeugt, sonst ein Knoten (`TiNRule` *Operatorname Operatordefinition*):

```
tiAllocateSc :: TiHeap -> IfrsOpDef -> TiState ->
                                (TiHeap, (TiName, HeapAddress))
tiAllocateSc heap od env =
 let op = (odGetOp od env); name = (opName op env) in
  if opArity op env == [] then
    let (heap', addr) = heapInsert heap
                                (TiNCAF name od Nothing)
          in
      (heap', (name, addr))
  else
    let (heap', addr) = heapInsert heap (TiNRule name od)
          in
      (heap', (name, addr))
```

Somit können wir den initialen Zustand der Template Instantiation Machine erzeugen.

6.2.4.2 Der Evaluator

Jetzt gilt es, vom Startzustand ausgehend, den `main`-Term auszuwerten. Da bei der Auswertung Fehler auftreten können, z.B. falls kein Redex gefunden werden kann, kapseln wir die Ausgabe mit dem Fehler-Datentyp `TiResult`. Wir erwarten dabei, dass bereits der Eingabezustand gekapselt ist (dies kann auch dazu benutzt werden, in `tiStart` den Fehler abzufangen, dass kein `main`-Operator definiert wurde), so dass unsere Auswertungsfunktion folgenden Typ hat:

```
tiEval :: (TiResult TiState) -> [(TiResult TiState)]
```

Wie beschrieben berechnen wir die Liste der Zustände für jeden Reduktionsschritt, dabei gehen wir wie folgt vor: Zunächst muss geprüft werden, ob der aktuelle Zustand bereits der Endzustand ist. Falls nein, berechnen wir solange den nächsten Zustand, bis dieser der Endzustand ist:

```
tiEval state = state:rest_states
     where
     rest_states | tiFinal state = []
                 | otherwise = tiEval next_state
     next_state = tiDoAdmin (tiStep state)
```

Außerdem kann mittels der Funktion `tiDoAdmin` noch die Statistik-Einstellung verwaltet werden; da wir allerdings nur einen Zähler für die Reduktionsschritte erhöhen, soll dies hier nicht weiter vertieft werden. Wir befassen uns daher zunächst mit der Funktion `tiFinal`, die überprüft, ob der Endzustand erreicht ist, und anschließend mit der Funktion `tiStep`, die den nächsten Zustand berechnet.

Prüfung auf Erreichen des Endzustandes:

Die Funktion `tiFinal` überprüft zunächst, ob bei der Auswertung ein Fehler aufgetreten ist. Falls nein, wird der Zustand aus dem Fehler-Datentyp extrahiert und an die Funktion `tiFinal1` übergeben:

```
tiFinal :: (TiResult TiState) -> Bool
tiFinal res
  | errResultOk res = tiFinal1 (errGetResult res)
  | otherwise = True
```

Die Funktion `tiFinal1` stellt sicher, dass genau ein Element auf dem Stack liegt (da von der Konzeption bzgl. Auswertung in GDSOS-kompatiblen Reduktionssystemen her weitere Elemente auf dem Stack bedeuten würden, dass gerade eine Unterauswertung statt findet – wie bereits erwähnt werden wir aber außer der ursprünglichen Adresse keine weiteren Adressen mehr auf den Stack legen, da dies bei einem Vergleich mit dem Reduktionskontext nur stört). Dann wird die Term-Adresse aus dem Stack gelesen. Da wir Adressen zusammen mit dem Zustand ja zu einer Instanz der Termklasse gemacht haben, können wir nun eine noch zu definierende Funktion `ifrsTermIsValue` aufrufen, die für beliebige Terme in beliebigen Reduktionssystemen herausfindet, ob es sich um Werte handelt:

```
tiFinal1
 st@(TiState stack heap stats global contexts domains opdefs)
  | (stackOneLeft stack) =
              ifrsTermIsValue (tiGetStackAddr st) () st
  | otherwise = False
```

Widmen wir uns also der Definition der Funktion `ifrsTermIsValue`. Wie bereits angedeutet, benutzen wir dabei die Termklasse und außerdem die Reduktionssystem-Klasse, wir greifen also nicht direkt auf die Datentypen zu, sondern benutzen

ausschließlich die von diesen Klassen bereitgestellten Methoden. Die Funktion `ifrsTermIsValue` hat somit folgenden Typ:

```
ifrsTermIsValue ::
  (Term t env v1 op1,
   RedSys redsys env rg cs c ds d ods od
    cn cts ct cv dn doms dom dv rs r mt op mv v) =>
      t -> redsys -> env -> Bool
```

Bei GDSOS-kompatiblen Reduktionssystemen ist die Überprüfung, ob ein Term ein Wert ist, relativ einfach: Gemäß Definition 5.2.3. muss nämlich nur geprüft werden, ob der äußere Operator kanonisch ist und falls ja, ob alle Argumente an strikten Positionen ebenfalls Werte sind. Da wir allerdings auch erlaubt haben, zusätzlich beliebige Domains (anhand des Schlüsselwortes `value`) als Wertemengen zu definieren (um so z.B. die Antworten im Call-by-Need Lambda-Kalkül darzustellen – d.h. im Zusammenhang mit der Auswertung meinen wir mit Werten nicht kopierbare Ausdrücke, sondern die Ergebnisse der Auswertung), benötigen wir zusätzlich noch eine Matching-Funktion `ifrsDomIsMatchEx`, die überprüft, ob ein Term in einer Domain enthalten ist und müssen diese Überprüfung für sämtliche `value`-Domains durchführen. Die Implementierung der Funktion `ifrsTermIsValue` gestaltet sich somit wie folgt:

```
ifrsTermIsValue t rs st
 | trmIsOpTerm t st =
    let op = trmGetOp t st
      strict = opStrict op st
      args = trmGetOpTerms t st
      strictvalues =
        and (map ((\x -> ifrsTermIsValue x rs st) . snd )
         (filter ((`elem` strict) . fst) (zip [1..] args)))
           in
    if opCanonical op st then
      if strictvalues then
        True
      else
        ifrsMatchValueDomains1 op t rs st
    else
      ifrsMatchValueDomains1 op t rs st

 | otherwise = False
```

Wie man sieht, wird zunächst sichergestellt, dass der Term ein Operatorterm ist. Falls der Operator dann kanonisch ist, werden die strikten Argumente untersucht. Dabei werden in der lokalen Definition

```
strictvalues =
 and (map ((\x -> ifrsTermIsValue x rs st) . snd )
  (filter ((`elem` strict) . fst) (zip [1..] args)))
```

die Argumente mit ihrer Position markiert und anschließend mit der Haskell-Funktion `filter` so gefiltert, dass nur die strikten Positionen übrigbleiben. Für die restlichen Argumente wird dann jeweils rekursiv überprüft, ob es sich um Werte handelt.

Falls der Operator nicht kanonisch ist oder nicht alle Argumente an strikten Positionen Werte sind, wird der Term mit den `value`-Domains verglichen. Die Funktion `ifrsMatchValueDomains1` definieren wir dazu wie folgt:

```
ifrsMatchValueDomains1 op t rs st =
  let ds = domListVOp (rsDomains rs st) (opName op st) st
      test t rs st = (\x -> ifrsDomIsMatchEx t x rs st)
        in
    any (test t rs st) ds
```

Es wird dabei einfach die Funktion `ifrsDomIsMatchEx` auf alle Elemente der Liste der `value`-Domains angewendet und anhand der Haskell-Funktion `any` geprüft, ob für ein beliebiges Element der Liste das Matching erfolgreich war. Falls ja, so ist der Term also in einer `value`-Domain enthalten und somit ein Wert. Die Funktion `domListVOp`, die die `value`-Domainliste zusammenstellt, ist eine Methode der Klasse `Domains` und als

```
filter (\x -> domIsValue x env) (domList ds env)
```

vordefiniert.

Die Implementierung der Matching-Funktion `ifrsDomIsMatchEx` ist aufgrund der flexiblen Gestaltung der Domain-Definitionen (mit der Möglichkeit auf andere Domains zu verweisen oder Kontexte zu benutzen) sehr umfangreich und aufwendig. Dabei werden die Alternativen der übergebenen Domain-Definition nacheinander geprüft bis evtl. eine Übereinstimmung mit dem übergebenen Term gefunden wird. Für den Vergleich eines Domaintermes mit dem übergebenen Term ist dazu ein schrittweiser Vergleich der benutzten Operatoren und Operanden nötig, wobei die Domainvariablen natürlich beliebige Terme enthalten können. Bei der Benutzung von Kontexten oder Verweisen auf andere Domains in einem Domainterm müssen wir dann die entsprechende Definition laden und wiederum deren Struktur untersuchen, wobei wieder sämtliche Alternativen durchlaufen werden müssen, bis man eine Übereinstimmung findet.

Wir werden später noch ein Matching mit Kontexten betrachten, es handelt sich dabei um einen Spezialfall des *linearen Kontext-Matching-Problems* wie z.B. in [SSS01/03] diskutiert. In unserem Fall dürfen die Kontexte nicht einfach beliebige Terme mit einem Loch sein, sondern müssen aus einer Kontextdefinition hervorgehen, außerdem müssen wir Informationen über die Lochposition zurückliefern; wir werden diesen Prozess im Folgenden einfach als *Kontext-Matching* bezeichnen. Da das Vorgehen bei Domains analog ist, außer dass keine Löcher enthalten sind und somit keine Information über die Lochposition im Term benötigt wird (was das Verfahren gegenüber dem Kontext-Matching bereits deutlich vereinfacht), verzichten wir auf eine detaillierte Betrachtung des Domain-Matchings.

Berechnung des nachfolgenden Zustandes:

Die Funktion `tiStep` bekommt den aktuellen Zustand übergeben und berechnet daraus den nächsten Zustand. Aufgrund der Kapselung durch den Fehler-Datentyp `TiResult` hat die Funktion folgenden Typ:

```
tiStep :: TiResult TiState -> TiResult TiState
```

In der Implementierung dieser Funktion wird nun die obere Adresse im Stack nachgeschlagen und für den dazugehörigen Term ein möglicher Redex gesucht. Dies erledigen wir anhand einer Funktion `ifrsGetRedContextHoleTerms`, die das Matching des Terms mit dem Reduktionskontext vornimmt, d.h. für einen Term t werden alle Zerlegungen $t = R_t[t']$, mit $R_t \in R$ und R ist der Reduktionskontext, gefunden. Wir bezeichnen in diesem Zusammenhang t' als Lochterm. Aus allen Lochtermen werden von `ifrsGetRedContextHoleTerms` dann diejenigen herausgefiltert und zurückgeliefert, die einen Redex darstellen könnten, d.h. ein Lochterm darf selbst kein Wert sein, aber alle Operanden an strikten Positionen müssen Werte sein. Somit steht uns die Liste möglicher Redexe zur Verfügung. Es ist jedoch nicht so, dass diese Liste bei einem Reduktionskalkül mit eindeutiger Auswertungsreihenfolge höchstens ein Element hat: Wie in Abschnitt 4.3.1 gesehen, gibt es z.B. im Call-by-Need Lambda-Kalkül Terme, für die mehrere mögliche Redexe gefunden werden. Allerdings gibt es unter diesen möglichen Redexen nur einen, auf den dann tatsächlich eine Reduktionsregel angewandt werden kann.

Somit versuchen wir nacheinander die möglichen Redexe zu reduzieren, bis uns dies tatsächlich geglückt ist oder kein möglicher Redex mehr übrig ist. Die Ausführung des Reduktionsschrittes übernimmt dabei die Funktion `tiDispatch`, die die Adresse eines möglichen Redexes übergeben bekommt und einen Fehler zurückliefert, falls die Reduktion nicht möglich ist. Da wir Fehler durch den Typ `TiResult` gekapselt haben, können wir dann überprüfen, ob die Reduktion geglückt ist oder es ggf. mit dem nächsten Redex versuchen.

Bevor wir uns mit den Details der Funktion `tiDispatch` beschäftigen, werfen wir aber noch einen Blick auf die Funktion `ifrsGetRedContextHoleTerms`, die die Liste der möglichen Redexe zurückliefert. Für das Matching mit dem Reduktionskontext verwenden wir die Funktion `ifrsMatchContext`, die wir später im Zusammenhang mit dem Suchen der passenden Auswertungsregel betrachten wollen, da wir sie dort des öfteren benötigen, wenn Kontexte auf der linken Seite der Regeln benutzt werden. Diese liefert zusätzlich zu jedem Lochterm noch eine Datenstruktur zurück, aus der die Lochposition hervorgeht. Denn auch wenn wir Adressen als Terme benutzen und somit als Lochterm gleich dessen Adresse erhalten, könnte es aufgrund des Sharings sein, dass sich mehrere Unterterme an verschiedenen Positionen die Referenz auf den Lochterm teilen, so dass wir aus der Adresse nicht schließen können, welcher dieser Unterterme derjenige ist, der im Loch steht. Der Typ dieser Datenstruktur ist `IfrsCtxtTpl t`, wobei `t` der Typ des untersuchten Terms ist. Wir benötigen sie an dieser Stelle nicht, da wir uns nicht für die Position des Redexes innerhalb des auszuwertenden Terms interessieren. Wir lassen die Funktion `ifrsGetRedContextHoleTerms` aber trotzdem auch diesen Datentyp mit zurückliefern, da die Funktion so auch als Schnittstelle für andere Funktionen dienen kann,

bei denen die Position des möglichen Redexes interessant ist (z.B. wenn die Zerlegung $t = R_t[t']$ dargestellt werden soll). Der Typ dieser Funktion ist somit:

```
ifrsGetRedContextHoleTerms ::
  (Term t env v1 op1,
   RedSys redsys env rg cs c ds d ods od
            cn cts ct cv dn doms dom dv rs r mt op mv v) =>
    t -> redsys -> env -> [(IfrsCtxtTpl t,t)]
```
[42]

In der Funktion wird nun der Name des Reduktionkontextes aus dem übergebenen Reduktionsssystem gelesen und dann, wie beschrieben, das Matching ausgeführt und das Ergebnis gefiltert:

```
ifrsGetRedContextHoleTerms t rs st =
  let rg = rsGlobal rs st
      (_,rc) = rsGetRedContext rg st
      rcn = ctGetName rc st
      hs = ifrsMatchContext t rcn rs st Nothing True
      f = (\(y,x) -> ifrsTermIsReducible x rs st)
        in
     filter f hs
```

Die Funktion `ifrsTermIsReducible` ist dabei wie folgt definiert:

```
ifrsTermIsReducible ::
  (Term t env v1 op1,
   RedSys redsys env rg cs c ds d ods od
            cn cts ct cv dn doms dom dv rs r mt op mv v) =>
      t -> redsys -> env -> Bool
ifrsTermIsReducible t rs st
  | ifrsTermIsValue t rs st = False
  | trmIsOpTerm t st =
    let op = trmGetOp t st
        strict = opStrict op st
        args = trmGetOpTerms t st
        f = (\x -> (ifrsTermIsValue (args !! (x-1)) rs st))
         in
      and (map f strict)

  | otherwise = False
```

[42] Der Interpreter Hugs liefert an dieser Stelle (und ähnlichen Stellen) des Programms einen (Typ-)Fehler, wenn der Typ explizit angegeben wird, berechnet bei Auskommentierung der Typangabe aber einen Typ, der diesem, bis auf Umbenennung der Typ-Variablen, entspricht. Da der Interpreter GHCi des Haskell-Compilers GHC in diesem Fall keinen Fehler zurückliefert, ist das als Hugs-spezifischer Fehler in der Typ-Überprüfung bei Multi-Parameter-Klassen zu sehen.

Es wird also die bereits beschriebene Funktion `ifrsTermIsValue` benutzt, um zu überprüfen, ob der Term selbst kein Wert ist, dafür aber alle Operanden an strikten Positionen.

Wenden wir uns der angekündigten Funktion `tiDispatch` zu, die den aktuellen Zustand der Template Instantiation Machine und die Adresse eines möglichen Redexes übergeben bekommt und daraus den Folgezustand berechnet, sofern es eine Regel gibt, die auf den möglichen Redex angewandt werden kann. Der Typ dieser Funktion ist:

```
tiDispatch :: TiState -> HeapAddress -> TiResult TiState
```

Die Funktion macht eine Fallunterscheidung über den Heap-Knoten, der an der übergebenen Adresse gespeichert ist:

- Falls es sich um einen Applikations-Knoten handelt, so betrachten wir den Verweis auf den Knoten, der die Operatordefinition enthält. Handelt es sich dabei um einen `TiNRule`-Knoten, so suchen wir die passende Auswertungsregel und wenden sie an. Handelt es sich hingegen um einen `TiNCAF`-Knoten, so gehen wir, falls Sharing verwendet werden soll, wie folgt vor: Wir überprüfen zunächst, ob der Term bereits ausgewertet wurde. Falls ja, so überschreiben wir den Applikations-Knoten einfach mit einer Indirektion auf den `TiNCAF`-Knoten (der ja wiederum die Referenz auf den ausgewerteten Term enthält). Falls der Term hingegen noch nicht ausgewertet wurde, so rufen wir `tiDispatch` mit der Adresse des `TiNCAF`-Knotens auf und überschreiben den Applikations-Knoten dann mit einer Indirektion auf das Ergebnis dieses Aufrufes.

- Falls es sich um einen `TiNCAF`-Knoten handelt, so wird überprüft, ob bereits ein Verweis auf einen teilweise ausgewerteten Term vorhanden ist. Es kann sich an dieser Stelle nicht um einen vollständig ausgewerteten Term handeln, da die Funktion `tiDispatch` nicht für Werte aufgerufen wird. Ist nun ein solcher Verweis vorhanden, so erfolgt ein rekursiver Aufruf von `tiDispatch` mit der Verweis-Adresse. Falls jedoch kein solcher Verweis vorhanden ist, so muss wiederum die passende Regel gefunden und angewendet werden. Der `TiNCAF`-Knoten wird dann durch einen neuen `TiNCAF`-Knoten überschrieben, der dem alten entspricht, außer dass dieser einen Verweis auf das Ergebnis der Regelanwendung enthält.

- Handelt es sich um einen Indirektions-Knoten, so erfolgt ein rekursiver Aufruf von `tiDispatch` mit der Indirektions-Adresse.

- Sonstige Knoten `TiNAbs` oder `TiNVar` dürfen nur als Operanden verwendet werden und führen somit zu einem Fehler.

Betrachten wir den Fall genauer, dass `tiDispatch` an der übergebenen Adresse `a` einen Heap-Knoten `TiNAp n addr addrs` vorfindet und unter der Adresse `addr` einen Knoten `TiNRule n' od`. Das Suchen und Anwenden der passenden Reduktionsregel gestaltet sich wie folgt:

```
do
  rule <- tiGetRule st od addrs
  return (tiInstantiateAndUpdate rule st a)
```

Die Funktion `tiGetRule` sucht dabei die Reduktionsregel und liefert einen Fehler zurück, falls keine passende Regel gefunden werden kann. Der Fehler wird dabei durch die Fehler-Monade abgefangen. Falls jedoch die Suche nach einer Reduktionsregel erfolgreich war, so wird diese nun angewendet. Dabei instanziiert die Funktion `tiInstantiateAndUpdate` die rechte Seite der Regel und überschreibt dann den Applikationsknoten an der Adresse `a` durch eine Indirektion, die auf das Ergebnis der Instanziierung verweist. Sowohl beim Suchen der Regel als auch bei der Instanziierung müssen allerdings einige Sachen beachtet werden: Um die rechte Seite einer Regel instanziieren zu können, müssen auch die Adressen der Argumente auf der linken Seite der Regel zur Verfügung stehen. Die Funktion `tiGetRule` soll daher zusammen mit der Regel auch die benötigten Adressen zurückliefern. Bei Kontext-Benutzungen reicht allerdings aufgrund des Sharings die Adresse alleine nicht aus, wir benötigen zusätzlich die bereits im Zusammenhang mit der Funktion `ifrsGetRedContextHoleTerms` angedeutete Datenstruktur `IfrsCtxtTpl`, um die Loch-Position bestimmen zu können. Bei der Instanziierung der Regel besteht eine besondere Schwierigkeit darin, sicherzustellen, dass die in Definition 5.1.9. vorausgesetzten Bedingungen sichere Regel und sichere Instanz erfüllt sind, d.h. wir müssen unerwünschtes Einfangen von Variablen durch Umbenennung verhindern.

Da also sowohl das Suchen als auch das Anwenden der Regel sehr umfangreich ausfällt, widmen wir diesen beiden Funktionalitäten jeweils einen eigenen Unterabschnitt.

Suchen der Reduktionsregel

Die Funktion `tiGetRule` bekommt den aktuellen Zustand, die Operatordefinition aus dem Regel-Knoten und die Argument-Adressen aus dem Applikations-Knoten übergeben und sucht nun die passende Regel. Neben der Regel werden auch gleich Informationen über die Argumente zurückgeliefert. Für jede Metavariable, Domain und für jeden Kontext auf der linken Seite einer Regel speichern wir somit in einer Assoziationsliste die Adressen der entsprechenden Instanzen, außerdem bei Abstraktionen die Namen der gebundenen Variablen in der Regel und in der Instanz. Für Kontexte speichern wir zusätzlich eine Datenstruktur vom Typ `IfrsCtxtTpl HeapAddress`, die den Kontext, bzw. die Lochposition beschreibt, außerdem wird für Kontext-Benutzungen der Form $x.C[x]$ der Name der im Loch gespeicherten Variable gespeichert (wir benutzen hier einen Listentyp, aber da keine Mehrfach-Loch-Kontexte erlaubt sind und somit immer nur eine Variable auf diese Weise benutzt werden kann, könnte man auch den Typ Maybe verwenden). Zudem ist es möglich, dass Kontexte nichtdeterministisch definiert sind und somit möglicherweise mehrere Lochpositionen gefunden werden. Bei Kontext-Benutzungen der Form $C[X]$, wobei X eine Metavariable ist, müssten somit auch mehrere Adressen für X betrachtet werden. Wir speichern also bei Metavariablen immer eine Liste von Adressen, außerdem speichern wir bei Kontexten eine Liste des Tupels, das die Datenstruktur `IfrsCtxtTpl` und die Adresse der Kontext-Instanz enthält. Der Typ der Funktion `tiGetRule` lautet wie folgt:

```
tiGetRule :: TiState -> IfrsOpDef -> [HeapAddress] ->
 TiResult
  (IfrsRule,
    -- Informationen über Metavariablen:
   (Assoc String ([HeapAddress], ([String],[String]))),
    -- Informationen über Kontexte:
   (Assoc String ((HeapAddress,
     [((IfrsCtxtTpl HeapAddress),HeapAddress)]),
     [String],([String],[String]))),
    -- Informationen über Domains:
   (Assoc String (HeapAddress,([String],[String]))))
```

Die Funktion `tiGetRule` ruft eine Funktion `tiGetRules` auf, die die Liste der anwendbaren Reduktionsregeln zurückliefert, da die Regeln ja nichtdeterministisch definiert sein könnten. Anschließend wird sichergestellt, dass genau eine Regel gefunden wurde (ansonsten wird ein Fehler zurückgeliefert) und dass bei Kontexten genau eine Lochposition gefunden wurde, d.h. die oben beschriebenen Listen, die die Positionen enthalten, haben genau ein Element.

Die Funktion `tiGetRules` schlägt nun die Regeln in der Operatordefinition nach und ruft eine Hilfsfunktion `tiGetRules1` auf, die als zusätzliches Argument `False` übergeben bekommt:

```
tiGetRules st od addrs =
  let rules = rulesOp (odGetRules od st) addrs st in
    tiGetRules1 st rules addrs False
```

Der zusätzliche Parameter in der Funktion `tiGetRules1` wird benötigt, um sich zu merken, ob bereits eine passende nichtdeterministische Regel gefunden wurde. Falls dieser Parameter `True` ist, werden nämlich nur noch weitere nichtdeterministische Regeln betrachtet. Um nun zu überprüfen, ob die nächste Regel in der übergebenen Liste der Regeln zu den Argumenten passt, deren Adressen in der übergebenen Adressenliste gespeichert sind, werden die Metaterme der linken Seite der Regel mit den Termen an den Argument-Adressen verglichen. Um nacheinander je einen Metaterm und einen Term vergleichen zu können, werden in einer lokalen `let`-Definition die Metaterme und die Term-Adressen zu einer Liste aus Metaterm-Term-Paaren „gezippt“:

```
zrterms = zip (ruleGetArgs r st) addrs
```

Den Vergleich übernimmt eine Funktion `tiMatchRuleTerms`:

```
(resflag,mvars,ctxts,doms) = tiMatchRuleTerms st zrterms
```

Das Rückgabe-Tupel enthält dabei ein Flag, das angibt, ob die Metaterme der Regel zu den Instanzen passen, außerdem die Assoziationslisten für Metavariablen, Kontexte und Domains, die die benötigten Informationen (Adressen usw.) über deren Instanzen enthalten, wobei die entsprechenden Assoziationslisten leer sind, falls das Matching fehlschlägt.

Falls also `resflag` gleich `True` ist, so fügen wir

```
res = (r,mvars,ctxts,doms)
```

in die Ergebnisliste ein, bzw. falls `r` deterministisch ist, liefern wir `[res]` zurück.

Betrachten wir somit die Funktion `tiMatchRuleTerms` genauer:

Falls die übergebene Liste der Metaterm-Term-Paare bereits leer ist, so liefern wir

```
(True, assocEmpty, assocEmpty, assocEmpty)
```

zurück. Ansonsten betrachten wir das erste Element dieser Liste mittels der Funktion `tiMatchRuleTerm`, die also einen Metaterm mit einem Term vergleicht. Falls das Matching erfolgreich ist, so wird rekursiv `tiMatchRuleTerms` mit der Restliste aufgerufen. Falls dabei alle Vergleiche zutreffen, d.h. das Flag in dem zurückgelieferten Tupel ist `True`, so werden die Assoziationslisten der Aufrufe von `tiMatchRuleTerm` und `tiMatchRuleTerms` mit der Restliste vereinigt und das Ergebnis zurückgeliefert. Ansonsten wird

```
(False, assocEmpty, assocEmpty, assocEmpty)
```

zurückgeliefert.

Der besonders interessante Teil der Regelsuche ist nun das Matching eines Metaterms mit einem Term in der Funktion `tiMatchRuleTerm`. Dazu wird zunächst der Heap-Knoten an der übergebenen Term-Adresse betrachtet, falls es sich um eine Indirektion handelt, wird `tiMatchRuleTerm` rekursiv mit der Indirektions-Adresse aufgerufen. Ansonsten betrachten wir den Metaterm:

Bei einer Metavariable ist das Matching erfolgreich und wir liefern

```
(True,[(name,([addr],([],[])))],[],[])
```

zurück, wobei `name` der Name der Metavariablen ist und `addr` die Adresse des Terms.

Bei einer Abstraktion sind drei Fälle zu unterscheiden:

1. Die Abstraktion enthält eine Meta-Applikation, bei der die gebundenen Variablen eingesetzt werden, d.h. sie hat folgende Form: $x\,y.X\{x, y\}$. Dann liefern wir

```
(True,[(name,([addr],(rvnames,tvnames)))],[],[])
```

zurück, wobei `name` der Name der Metavariablen ist, `addr` die Adresse des Terms und `rvnames` bzw. `tvnames` sind die Namen der abstrahierten Variablen im Metaterm bzw. Term.

2. Die Abstraktion enthält eine Kontext-Benutzung, wobei der Lochterm die gebundene Variable ist, d.h. sie hat folgende Form: $x.C[x]$. Es ist nun ein Matching des Kontextes C mit dem Term innerhalb der Abstraktion erforderlich, wobei zusätzlich noch sichergestellt

werden muss, dass der Lochterm die abstrahierte Variable enthält. Der Term innerhalb der Abstraktion ist dabei der Term, der bei einem Abstraktions-Knoten `(TiNAbs vaddrs taddr)` an der Adresse `taddr` zu finden ist. Das Kontext-Matching übernimmt die bereits im Zusammenhang mit der Redex-Suche angedeutete Funktion `ifrsMatchContext`, die wir anschließend in einem eigenem Unterabschnitt betrachten. Der Aufruf gestaltet sich dabei innerhalb einer lokalen `let`-Definition wie folgt:

```
hlist = (ifrsMatchContext taddr cname () st
                                  (Just (head tvnames)) False)
```

Hierbei ist `taddr` die Adresse des Terms innerhalb der Abstraktion, `cname` der Name des Kontextes, `st` der aktuelle Zustand und `tvnames` sind die Namen der abstrahierten Variablen innerhalb des Terms. Wir übergeben also der Funktion, neben dem zu matchenden Term und dem Kontextnamen, den Namen der Variablen, die sich im Loch befindet, und ein zusätzliches Argument `False`, welches besagt, dass keine spezielle nichtdeterministische Behandlung sämtlicher Kontexte (wie bei der Redex-Suche) notwendig ist. Die Funktion `ifrsMatchContext` liefert nun die gewünschten Informationen über die Loch-Positionen zurück. Wir müssen uns also nur noch davon überzeugen, dass diese Liste nicht leer ist:

```
case hlist of
 [] -> (False,[],[],[])
 (xs) ->
     (True,[],[(name,((taddr,xs),[],(rvnames,tvnames)))],[])
```

3. Die Abstraktion enthält eine Domain, d.h. sie hat die Form *x.D*. Wir müssen somit anhand der Funktion `ifrsDomIsMatch` überprüfen, ob der Term innerhalb der Abstraktion in der Domain liegt:

```
if (ifrsDomIsMatch taddr dname () st) then
     (True,[],[],[(dnamestr,(taddr,(rvnames,tvnames)))])
else
     (False,[],[],[])
```

Dabei ist `dname` (mit Position) und `dnamestr` (ohne Position) der Domainname, `taddr` die Adresse des Terms innerhalb der Abstraktion und `rvnames` bzw. `tvnames` sind die Namen der abstrahierten Variablen im Metaterm bzw. Term.

Es sind bei der Betrachtung des übergebenen Metaterms noch drei weitere Fälle übrig, es könnte sich noch um einen Operatorterm, eine Kontextbenutzung mit einer Metavariable oder um eine Domain handeln.

Falls wir es bei unserem Metaterm mit einem Operatorterm zu tun haben, so stellen wir sicher, dass auch der Term ein Operatorterm ist, und vergleichen dann einfach rekursiv die Operanden des Terms mit denen des Metaterms.

Bei einer Kontextbenutzung mit einer Metavariablen benutzen wir wieder die Funktion `ifrsMatchContext` für das Kontext-Matching. In einer lokalen let-Definition definieren wir:

```
hlist = (ifrsMatchContext addr cname () st Nothing False)
```

Hierbei ist `addr` die Adresse des zu matchenden Terms, `cname` der Name des Kontextes und `st` der aktuelle Zustand. Da wir nicht, wie bei einer Abstraktion der Form $x.C[x]$, noch sicherstellen müssen, dass sich eine bestimmte Variable im Loch befindet, übergeben wir hier statt eines Variablennamens `Nothing` und wiederum als weiteres Argument `False`, da keine Spezialbehandlung wie bei der Redex-Suche nötig ist.

Wie gewohnt müssen wir nun sicherstellen, dass mindestens eine Lochposition gefunden wurde:

```
case hlist of
 [] -> (False,[],[],[])
 (xs) ->
  let xs1 = map snd xs in
   (True,[(mvarname,(xs1,([],[])))],
         [(name,((addr,xs),[mvarname],([],[])))],[])
```

Dabei ist `mvarname` der Name der Metavariablen.

Der letzte übriggebliebene Fall ist, dass die Metavariable auf eine Domain verweist. In diesem Fall prüfen wir einfach anhand der Funktion `ifrsDomIsMatch`, ob der Term in der gewünschten Domain liegt:

```
if (ifrsDomIsMatch addr dname () st) then
     (True,[],[],[(dnamestr,(addr,([],[])))])
else
     (False,[],[],[])
```

Hierbei ist `dname` (mit Position) bzw. `dnamestr` (ohne Position) der Domain-Name und `addr` die Adresse des Terms.

Die Betrachtung der Funktion `tiMatchRuleTerm` ist hiermit abgeschlossen und wir sind nun in der Lage, die passende Reduktionsregel zu finden, wenn uns die Funktionen für Kontext- und Domain-Matching zur Verfügung stehen. Wir betrachten nur das Kontext-Matching, da das Domain-Matching analog abläuft, außer dass die Betrachtung der Lochpositionen entfällt und wir lediglich `True` (Match) oder `False` (kein Match) zurückliefern müssen.

Kontext-Matching

Wie bereits erwähnt, ist unser Vorgehen, um für einen Term eine Zerlegung in einen Kontext und einen Lochterm zu finden, ein Spezialfall des linearen Kontext-Matching-Problems. In [SSS01/03] werden dabei Kontext-Variablen verwendet, die an beliebiger Stelle im Termbaum vorkommen und daher Kontext-Unterterme enthalten dürfen. Somit entsprechen sie bei unserem Kontextterm-Begriff den Kontext-Benutzungen, jedoch mit dem Unterschied, dass diese Kontext-Variablen zu beliebigen Kontexten instanziiert werden können, während wir für unsere Kontext-Benutzungen einen Kontext finden müssen, der in der gewünschten Kontext-(Mengen-)Definition enthalten ist. Außerdem

verwenden wir den Begriff Kontext-Variablen hier für gewöhnliche Variablen in Kontexttermen, die beliebige Terme enthalten können. Wir wollen sie durch diesen Begriff von den Variablen in (normalen) Termen abgrenzen und natürlich auch von den Metavariablen in den Metatermen. Den in [SSS01/03, Abschnitt 4] angegebenen Algorithmus müssen wir nun so abändern, dass wir für einen Kontext nacheinander die Kontext-Alternativen in der entsprechenden Kontext-Definition durchprobieren. Für jede Kontext-Alternative können wir dann wieder das angegebene Verfahren anwenden, bis wir auf die nächste Kontext-Benutzung oder Kontext-Referenz stoßen.

Da wir als Rückgabe auch eine detaillierte Information über den gefundenen Kontext und dessen Lochposition benötigen, definieren wir zunächst eine Datenstruktur, die diesen Kontext beschreibt. Da wir bei Kontext-Variablen nicht den matchenden Unterterm weiter betrachten wollen, enthält die Datenstruktur einen Konstruktor `IfrsCRef`, der einfach den gesamten Unterterm speichert. Ansonsten enthält die Datenstruktur die für die Darstellung von Termen benötigten Bestandteile Operatorterme, Abstraktionen und Variablen, außerdem natürlich einen Loch-Konstruktor. Die Definition der Datenstruktur gestaltet sich somit wie folgt:

```
data IfrsCtxtTpl a =
      IfrsCEmpty
    | IfrsCRef a
    | IfrsCVar String
    | IfrsCAbstraction [String] (IfrsCtxtTpl a)
    | IfrsCOpTerm String [(IfrsCtxtTpl a)]
  deriving (Eq)
```

Die Funktion `ifrsMatchContext` bekommt nun einen Term, den Namen eines Kontextes, das Reduktionssystem und die Umgebung bzw. den Zustand übergeben, außerdem `Just` *Variablenname*, falls sich eine bestimmte Variable im Loch befinden muss (sonst `Nothing`) und ein Flag, dass angibt, ob speziell für eine Redex-Suche auch die deterministischen Kontexte nichtdeterministisch behandelt werden sollen oder nicht. Die Funktion liefert dann eine Liste zurück, die die Lochterme enthält, sowie die eben beschriebene Datenstruktur, die den Kontext und dessen Lochposition beschreibt. Der Typ dieser Funktion ist somit:

```
ifrsMatchContext ::
  (Term t env v1 op1,
   RedSys redsys env rg cs c ds d ods od
            cn cts ct cv dn doms dom dv rs r mt op mv v) =>
     t -> cn -> redsys -> env -> Maybe String -> Bool ->
      [(IfrsCtxtTpl t,t)]
```

Bei der Implementierung der Funktion greifen wir wiederum auf eine Funktion `ifrsMatchContextRec` zurück, die speziell an die Bedürfnisse für rekursive Aufrufe angepasst ist:

```
ifrsMatchContext t cn rs st mbv rcflag =
  case ifrsMatchContextRec t cn rs st Nothing mbv rcflag of
    Right x -> x
    _ -> []
```

Das zusätzliche Argument, das wir mit `Nothing` initialisieren, kann dabei einen Kontextterm enthalten, dieser muss dann wiederum mit dem Lochterm matchen. Wir benötigen dies, da wir in Kontextdefinitionen selbst ja auch Kontext-Benutzungen erlaubt haben. Wir wollen die Funktion `ifrsMatchContextRec` auch zum Matching der Operanden verwenden, die kein Loch enthalten. Somit benutzen wir den Datentyp `Either` um das Ergebnis zurückzuliefern: Falls wir das Loch bzw. in einer nichtdeterministischen Definition mehrere Alternativen für Löcher finden und das Matching erfolgreich ist, liefern wir `Right` *x* zurück, wobei *x* die gewünschte Ergebnisliste mit den Lochtermen und den Kontextbeschreibungen durch die Datenstruktur `IfrsCtxtTpl` ist. Falls wir kein Loch finden, aber das Matching dieses Unterterms ohne Loch erfolgreich ist, liefern wir `Left Just` *c* zurück, wobei *c* die Beschreibung des Unterterms durch die Datenstruktur `IfrsCtxtTpl` ist. Falls das Matching fehlschlägt, liefern wir `Left Nothing` zurück. Beim initialen Aufruf der Funktion müssen wir also sicherstellen, das `Right x` zurückgeliefert wird, da sonst kein Loch gefunden wurde bzw. das Matching fehlgeschlagen ist. Der Typ der Funktion `ifrsMatchContextRec` lautet also wie folgt:

```
ifrsMatchContextRec ::
  (Term t env v1 op1,
   RedSys redsys env rg cs c ds d ods od
            cn cts ct cv dn doms dom dv rs r mt op mv v) =>
     t -> cn -> redsys -> env ->
     Maybe ct -> Maybe String -> Bool ->
      Either (Maybe (IfrsCtxtTpl t)) [(IfrsCtxtTpl t,t)]
```

In der Implementierung dieser Funktion wird zunächst anhand des Kontextnamens die passende Kontextdefinition im Reduktionssystem gesucht, dazu benutzen wir einfach die entsprechende Methode der Klasse `Contexts`. Die entsprechenden Kontextalternativen bzw. -terme übergeben wir dann an eine Funktion `ifrsMatchContextTerms`, die diese der Reihe nach testet.

Die Funktion `ifrsMatchContextTerms` liefert dabei einfach `Right []` zurück, falls die übergebene Liste der Kontextterme leer ist, und untersucht ansonsten das erste Element anhand einer Funktion `ifrsMatchContextTerm`. Falls das Matching erfolgreich ist, d.h. `ifrsMatchContextTerm` liefert `Right` *x* zurück und *x* ist nicht die leere Liste, so überprüfen wir je nach Determinismus-Einstellung auch noch die Restliste oder liefern das Ergebnis zurück. Falls das Matching nicht erfolgreich ist, liefern wir `Left Nothing` zurück.

Die Funktion `ifrsMatchContextTerm`, die einen Kontextterm mit einem Term vergleicht, ist also nun der zentrale Bestandteil des Kontextmatchings (analog zur Regelsuche, bei der die Funktion `tiMatchRuleTerm`, die einen Metaterm mit einem Term vergleicht, die entscheidende Rolle spielt). Zusätzlich zu den bisher beschriebenen

Parametern benötigen wir für gebundene Variablen noch eine Assoziationsliste, anhand derer wir die Variablennamen aus dem übergebenen Term den Namen der (Kontext-) Variablen aus dem Kontextterm zuordnen können.

Die Funktion `ifrsMatchContextTerm` hat somit folgenden Typ:

```
ifrsMatchContextTerm ::
  (Term t env v1 op1,
   RedSys redsys env rg cs c ds d ods od
            cn cts ct cv dn doms dom dv rs r mt op mv v) =>
    t -> ct -> redsys -> env ->
      Assoc String String ->
        Maybe ct -> Maybe String -> Bool ->
          Either (Maybe (IfrsCtxtTpl t)) [(IfrsCtxtTpl t,t)]
```

In der Implementierung der Funktion `ifrsMatchContextTerm` wird nun eine Fallunterscheidung über den übergebenen Kontextterm gemacht:

Bei einem Loch prüfen wir zunächst, ob im Rahmen einer Kontext-Benutzung innerhalb einer Kontext-Definition (bei einem rekursiven Aufruf) ein Kontextterm übergeben wurde, der mit dem Lochterm (in diesem Fall der übergebene Term, da wir ja bereits beim Loch angelangt sind) matchen muss: falls ja, so wird `ifrsMatchContextTerm` rekursiv mit diesem Kontextterm aufgerufen; falls nein, so könnte es aber noch sein, dass der Name einer Variable übergeben wurde, aus der der Lochterm bestehen muss: Dann lässt sich diese Bedingung zwar leicht mit Hilfe der Methoden der Term-Klasse prüfen, allerdings müssen wir hier zusätzlich noch darauf achten, dass die Variable nicht gebunden ist, also nicht in der übergebenen Assoziationsliste der gebundenen Variablen (auf der Term-Seite) enthalten ist. Dies ist deshalb von Bedeutung, weil bei einem Matching mit einem Metaterm der Form $x.C[x]$ z.B. für einen Term $\lambda(x.@(\lambda(x.x), x))$ keine Zerlegung $\lambda(x.@(\lambda(x.[x]), x))$ möglich sein soll. Sind an dieser Stelle die genannten Bedingungen erfüllt, so liefern wir `Right [(IfrsCEmpty,t)]`zurück, wobei `t` der übergebene Term ist, sonst `Left Nothing`. Ebenso liefern wir das Ergebnis `Right [(IfrsCEmpty,t)]`zurück, wenn keine weitere Prüfung des Lochterms mehr nötig ist.

Bei einer (Kontext-)Variablen prüfen wir, ob diese in der übergebenen Assoziationsliste der gebundenen Variablen enthalten ist. Falls ja, müssen wir dann sicherstellen, dass der Term genau die Variable enthält, die zu dieser Kontext-Variablen korrespondiert. Ist dies der Fall, so liefern wir `Left (Just (IfrsCVar bvt))` zurück, wobei `bvt` der Name der Variablen im Term ist, falls dies jedoch nicht erfüllt ist, liefern wir `Left Nothing` zurück. Sofern die Kontext-Variable jedoch nicht gebunden ist, muss nichts weiter geprüft werden und wir liefern `Left (Just (IfrsCRef t))` zurück, wobei `t` der übergebene Term ist.

Bei einer Referenz auf einen weiteren Kontext rufen wir einfach rekursiv die Funktion `ifrsMatchContextRec` mit dem neuen Kontextnamen auf.

Bei einer Kontext-Benutzung rufen wir ebenfalls `ifrsMatchContextRec` mit dem neuen Kontextnamen auf, übergeben aber zusätzlich den in das Loch eingesetzten

Kontextterm, so dass dann, wie oben beschrieben, bei einem Loch das Matching mit diesem Kontextterm fortgesetzt werden kann.

Bei einer Referenz auf eine Domain prüfen wir mit `ifrsDomIsMatch`, ob der Term in der Domain-Definition enthalten ist, falls ja, liefern wir `Left (Just (IfrsCRef t))` zurück, sonst `Left Nothing`.

Bei einer Abstraktion muss zunächst sichergestellt werden, dass es sich bei dem Term auch um eine Abstraktion handelt und die Anzahl der abstrahierten Variablen gleich ist (ansonsten wäre aber auch die Stelligkeit verletzt). Anschließend fügen wir diese abstrahierten Variablen in die Assoziationsliste der gebundenen Variablen ein; in einer lokalen `let`-Definition sieht das wie folgt aus:

```
nbv = (assocInsertUpdateList bv (zip ncvs nvs))
```

Hierbei ist `bv` die übergebene Assoziationsliste der gebunden Variablen, `ncvs` sind die Namen der abstrahierten Variablen im Kontextterm und `nvs` sind die entsprechenden Variablennamen im Term, das Ergebnis `nbv` ist dann die neue Assoziationsliste. Nun müssen wir rekursiv den Kontextterm und den Term innerhalb der Abstraktionen vergleichen:

```
rec = ifrsMatchContextTerm at cat rs st nbv usect mbv rcflag
```

Dabei ist `at` der Term innerhalb der (Term-)Abstraktion, `cat` der Kontextterm innerhalb der (Kontextterm-)Abstraktion, `nbv` die neue Assoziationliste und die restlichen Argumente werden unverändert weitergegeben. Jetzt müssen wir natürlich noch den Rückgabewert des rekursiven Aufrufs prüfen und, falls das Matching erfolgreich ist, noch die Information im Datentyp `IfrsCtxtTpl` um ein `IfrsCAbstraction` erweitern:

```
case rec of
  Right x ->
    (Right (map (\(y,z) -> ((IfrsCAbstraction nvs y),z)) x))
  Left (Just x) ->
    (Left (Just ((\y -> (IfrsCAbstraction nvs y)) x)))
  Left Nothing ->
    (Left Nothing)
```

Als letzter Fall bei der Fallunterscheidung über den übergebenen Kontextterm in unserer Funktion `ifrsMatchContextTerm` bleibt jetzt noch ein Operatorterm. Zunächst müssen wir überprüfen, ob auch der Term ein Operatorterm ist und ob der verwendete Operator gleich ist. Ist dies nicht der Fall, so liefern wir `Left Nothing` zurück, ansonsten müssen wir nun die Operanden prüfen. Dies erledigen wir, indem wir der Reihe nach (mit Hilfe von `map`) durch einen rekursiven Aufruf von `ifrsMatchContextTerm` die Operanden matchen und dann das Ergebnis anhand einer Funktion `ifrsCtMatchList` prüfen:

```
ifrsCtMatchList (opName op st)
 (map
  (\x -> ifrsMatchContextTerm (fst x) (snd x)
                              rs st bv usect mbv rcflag)
   (zip ot cot))
```

Hierbei ist `op` der Operator (es wird ja im Kontextterm und im Term der gleiche Operator verwendet), `st` die Umgebung bzw. der Zustand, `ot` sind die Operanden im Term, `cot` die Operanden im Kontextterm und die restlichen Argumente des rekursiven Aufrufs werden einfach unverändert weitergegeben.

Betrachten wir nun die Funktion `ifrsCtMatchList`. Diese bekommt den Operatornamen und die Ergebnisse der Matchings der Operanden übergeben und berechnet dann aus dieser Liste der Ergebnisse ein Gesamtergebnis:

```
ifrsCtMatchList :: String ->
  [Either (Maybe (IfrsCtxtTpl a)) [(IfrsCtxtTpl a,b)]] ->
    Either (Maybe (IfrsCtxtTpl a)) [(IfrsCtxtTpl a,b)]
```

Falls die übergebene Liste leer ist, so liefern wir einfach `Left (Just (IfrsCOpTerm opname []))` zurück.

Ansonsten betrachten wir das erste Element der Liste:

Ist dieses das Ergebnis eines erfolgreichen Matchings, das ein Loch enthält, d.h. das Ergebnis hat die Form `Right ys`, so wird zunächst rekursiv mit der Restliste weitergemacht und anschließend das Ergebnis betrachtet: Dieses muss die Form `Left Left (Just (IfrsCOpTerm _ tpls))` haben, da sonst entweder mehrere (Kontext-) Operanden Löcher enthalten oder das weitere Matching der Operanden fehlgeschlagen ist. Ist die Form erfüllt, so liefern wir

```
Right (map (\(y,z) -> ((IfrsCOpTerm opname (y:tpls)),z)) ys)
```

zurück, ansonsten `Left Nothing`.

Ist das nächste Element das Ergebnis eines erfolgreichen Matchings eines Terms mit einem Kontextterm ohne Loch, d.h. das Ergebnis hat die Form `Left (Just y)`, so wird ebenso rekursiv mit der Restliste weitergemacht. Es folgt eine Fallunterscheidung über das Ergebnis dieses rekursiven Aufrufs: Hat dieses die Form `Right ys`, so liefern wir

```
(Right (map (\((IfrsCOpTerm _ tpls),z) ->
                 ((IfrsCOpTerm opname (y:tpls)),z)) ys))
```

zurück, bei einer Form `Left (Just (IfrsCOpTerm _ tpls))` liefern wir hingegen

```
(Left (Just (IfrsCOpTerm opname (y:tpls))))
```

zurück. Ist jedoch bereits beim rekursiven Aufruf das Matching fehlgeschlagen, d.h. dieses Ergebnis hat die Form `Left Nothing`, so liefern wir auch `Left Nothing` zurück.

Als letzte Möglichkeit bei der Betrachtung des nächsten Elements der übergebenen Liste bleibt nun nur noch, dass dieses `Left Nothing` ist, und wir liefern somit auch hier wieder `Left Nothing` zurück.

Die Funktion `ifrsCtMatchList` stellt also im Wesentlichen nur sicher, dass in der übergebenen Liste mit den Ergebnissen der Matchings der Operanden genau ein Element die Form `Right ys` hat und die restlichen Elemente die Form `Left (Just (...))`. Etwas kompliziert ist dabei dann nur noch, dass `ys` eine Liste mit verschiedenen Alternativen für die Lochposition sein kann und wir daher die Operanden in die Struktur des `IfrsCtxtTpl`-Datentyps für jede dieser Alternativen einfügen müssen. Daher die etwas unübersichtlichen Rückgaben der Form `Right (map (...) ys)`.

Somit haben wir auch die Funktionen des Kontext-Matchings etwas näher betrachtet. Wie wir gesehen haben, muss dazu im Wesentlichen nur die Struktur der Kontextterme und der Terme durchlaufen werden - kompliziert wird es allerdings durch die Vielzahl der Definitions-Möglichkeiten, mit solchen Sonderfällen wie der Kontext-Benutzung innerhalb einer Kontext-Definition.

Anwenden der Regel

Wir haben nun eine Reduktionsregel gefunden, die auf unseren Term angewendet werden kann. Wie gesehen, liefert die Regelsuche ein Tupel `(r,mvars,ctxts,doms)` zurück, das neben der Regel `r` auch Assoziationslisten `mvars`, `ctxts` und `doms` enthält, in denen man die benötigten Informationen, wie z.B. Adressen der zugehörigen Instanzen, nachschlagen kann. Jetzt gilt es also, die rechte Seite der Regel mit Hilfe dieser Informationen zu instanziieren. Wir definieren dazu im Folgenden zwei Funktionen: eine Funktion `tiInstantiate`, die den instanziierten Term an einem neuen Speicherplatz im Heap ablegt (es sei denn es handelt sich bei diesem Term um ein bereits instanziiertes Argument, dann bleibt der Heap unverändert und es wird nur die Adresse des instanziierten Terms zurückgeliefert), und eine Funktion `tiInstantiateAndUpdate`, die den Heap-Knoten an einer gegebenen Adresse überschreibt. In beiden Fällen greifen wir dabei auf eine Hilfs-Funktion `tiInstantiate1` zurück.

Die Funktion `tiInstantiate` definieren wir einfach wie folgt:

```
tiInstantiate (r,mvars,ctxts,doms) st =
  let mt = ruleGetResult r st
      (heap, addr) =
          tiInstantiate1 mt st mvars ctxts doms assocEmpty
             in
    (st{tiHeap = heap},addr)
```

Wir extrahieren also zunächst den Metaterm auf der rechten Seite der Regel aus der Regeldefinition und übergeben diesen dann der Funktion `tiInstantiate1`, diese liefert ein Tupel, bestehend aus einem neuen Heap und der Adresse des neu erzeugten

Knotens bzw. des instanziierten Terms in diesem Heap, zurück. Wir ersetzen dann den Heap im aktuellen Zustand und liefern diesen zusammen mit der Adresse zurück.

Die Funktion `tiInstantiateAndUpdate` greift zunächst auf eine weitere Hilfs-Funktion `tiInstantiateAndUpdate1` zurück:

```
tiInstantiateAndUpdate (r,mvars,ctxts,doms) st addr =
  let mt = ruleGetResult r st
       in
    tiInstantiateAndUpdate1 mt st mvars
                            ctxts doms assocEmpty addr
```

Die Funktion `tiInstantiateAndUpdate1` macht eine Fallunterscheidung über den Metaterm:

```
tiInstantiateAndUpdate1 mt st mvars ctxts doms vars addr =
 let opdefs = tiOpDefs st
       in
    case mt of
```

Zunächst stellen wir sicher, dass es sich nicht um eine Abstraktion handelt, da Abstraktionen keine gültigen Terme sind und wir somit nie Terme mit einer Abstraktion überschreiben wollen:

```
      _ | mtIsAbstraction mt st ->
        error "Invalid attempt to update with abstraction!"
```

Falls es sich nun um einen Operatorterm handelt, so instanziieren wir zunächst anhand einer Funktion `tiMapInstantiate` alle Operanden, schlagen dann die Adresse des Heap-Knotens, der die Operatordefinition enthält, in der entsprechenden Assoziationsliste unseres Zustands nach und überschreiben den Heap-Knoten an der übergebenen Adresse durch einen Applikations-Knoten, der jetzt unseren Operatorterm darstellt:

```
      _ | mtIsOpTerm mt st ->
        let op = mtGetOp mt st
            terms = mtGetOpTerms mt st
            name = opName op st
            addr1 =
              (assocLookup opdefs name
                (error ("Undefined operator " ++ name)))
            (heap',addrs) =
              tiMapInstantiate st mvars ctxts doms vars terms
               in
          st{tiHeap = (heapUpdate heap' addr
                                  (TiNAp (name) addr1 addrs))}
```

In den übrigen Fällen rufen wir einfach `tiInstantiate1` auf und überschreiben den Heap-Knoten durch einen Indirektionsknoten auf das Ergebnis:

```
      _ ->
        let (heap', result_addr) =
              tiInstantiate1 mt st mvars ctxts doms vars
                in
          st{tiHeap = (heapUpdate heap' addr
                                    (TiNInd result_addr))}
```

Die Funktion `tiInstantiate1` ist also die Funktion, die die eigentliche Arbeit erledigt, denn auch die Funktion `tiMapInstantiate` ruft einfach `tiInstantiate1` für jedes Argument der übergebenen Liste auf. Hier ist nur noch darauf zu achten, dass stets der neue Heap weiterverwendet wird:

```
tiMapInstantiate st mvars ctxts doms vars [] =
  ((tiHeap st),[])

tiMapInstantiate st mvars ctxts doms vars (x:xs) =
  let heap = tiHeap st
      (heap1, x')  =
        tiInstantiate1 x st mvars ctxts doms vars
      tistate1 = st{tiHeap = heap1}
      (heap2, xs') =
        tiMapInstantiate tistate1 mvars ctxts doms vars xs
          in
    (heap2, x':xs')
```

Betrachten wir nun die Funktion `tiInstantiate1`. Diese hat folgenden Typ:

```
tiInstantiate1 :: IfrsMetaTerm -> TiState ->
   -- Informationen über Metavariablen:
  (Assoc String ([HeapAddress],([String],[String]))) ->
   -- Informationen über Kontexte:
  (Assoc String ((HeapAddress,
     [((IfrsCtxtTpl HeapAddress),HeapAddress)]),
     [String],([String],[String]))) ->
   -- Informationen über Domains:
  (Assoc String (HeapAddress,([String],[String]))) ->
   -- Informationen über gebundene Variablen:
  (Assoc String (String,HeapAddress)) ->
  (TiHeap, HeapAddress)
```

Wir benutzen dabei eine vierte Assoziationsliste für gebundene Variablen, die für rekursive Aufrufe bei Abstraktionen benötigt wird. In der Implementierung der Funktion `tiInstantiate1` wird auf gewohnte Art und Weise eine Fallunterscheidung über den Metaterm gemacht:

Ist der Metaterm eine Variable, so versuchen wir diese in der eben erwähnten Assoziationsliste der gebundenen Variablen zu finden. Ist die Variable nicht enthalten, so liefern wir einen Fehler zurück, da wir freie Variablen nicht instanziieren wollen. Ansonsten finden wir in der Assoziationsliste die Adresse `vaddr` des Heap-Knotens

dieser Variable und liefern einfach `(heap, vaddr)` zurück, wobei `heap` der unveränderte Heap ist.

Ist der Metaterm eine Metavariable, so suchen wir nun die Adresse `mvaddr` des dazugehörigen Terms in der Assoziationsliste der Metavariablen und liefern, analog zu den normalen Variablen, einfach `(heap, mvaddr)` zurück. Falls jedoch an dieser Stelle kein Sharing verwendet werden soll, müssen wir den Term kopieren. Dies erledigen wir mit Hilfe einer noch zu definierenden Funktion `tiCopyTree`.

Bei einer Abstraktion wird es etwas kniffliger, allerdings können wir auch hier einfach `(heap, mvaddr)` zurückliefern, falls eine Abstraktion der Form $x_1 \ldots x_n \,.\, X\{x_1,\ldots,x_n\}$ unverändert von der linken Seite der Regel übernommen wird. Wir müssen dazu nur sicherstellen, dass die gewünschten Variablennamen mit denen, die zusammen mit der Adresse der Metavariablen in der Assoziationsliste gespeichert wurden, übereinstimmen. Ebenso wäre es auch bei Abstraktionen der Form $x \,.\, C[x]$ oder $x_1 \ldots x_n \,.\, D$, wobei C ein Kontext und D eine Domain ist, möglich, diese unverändert von der linken Seite der Regel zu übernehmen, falls wir in den Assoziationslisten der Kontexte und Domains zusätzlich die Adressen der zugehörigen Abstraktionen speichern würden.

Ansonsten muss nun besonders Acht auf die gewünschten Bindungsvariablen gegeben werden: Da freie Variablen in den Instanzen der Metavariablen, Kontexte und Domains nicht eingefangen werden sollen, müssen die neuen Variablen, die den gleichen Namen wie eine dieser freien Variablen haben, umbenannt werden. Dazu bestimmen wir das kleinste n, so dass x_n eine noch nicht verwendete Variable ist. Die Namen der umzubenennenden Variablen ändern wir dann in x_n um, wobei wir n nach jeder Umbenennung um Eins erhöhen. Ist dies geschehen, so kann schließlich rekursiv der in der Abstraktion enthaltene Term instanziiert werden. Wir rufen an dieser Stelle einfach eine Funktion `tiInstantiateAbs1` auf, die die Umbenennungen und die weitere Instanziierung übernimmt; wir betrachten diese später noch etwas näher.

Auch bei einer Meta-Applikation müssen wir aufpassen, dass bei eingesetzten Termen mit freien Variablen diese nicht unerwünscht durch Bindungen innerhalb der Metavariablen, in die diese eingesetzt werden sollen, eingefangen werden. Beispielsweise müssen wir bei einer Meta-Applikation $Y\{x\}$, mit $y.Y\{y\} = y.@(\lambda(x.@(x, y)), \lambda(z.z))$, zu einer Variante $y.Y'\{y\} = y.@(\lambda(x'.@(x', y)), \lambda(z.z))$ wechseln, bei der wir x in x' umbenennen, damit die in der Meta-Applikation eingesetzte freie Variable x durch die Bindung $\lambda(x. \ldots)$ in Y eingefangen wird. Da wir die zu der Metavariablen, die für die Meta-Applikation verwendet wird, zugehörige Abstraktion kopieren müssen, bevor wir die Einsetzungen vornehmen können, erledigen wir die nötigen Umbenennungen einfach während des Kopiervorgangs. Wir verfahren somit wie folgt:

1. Zusammenstellen einer Liste der freien Variablen in den eingesetzten Termen,

2. Kopieren der Abstraktion, wobei bei jeder Bindungsvariablen geprüft wird, ob es eine freie Variable mit gleichem Namen gibt, falls ja wird die Bindungsvariable umbenannt,

3. und schließlich: Ersetzen der äußeren Bindungsvariablen durch die einzusetzenden Terme.

Beim Einsetzen der Terme brauchen wir dabei nur für einen Abstraktions-Knoten `TiNAbs` *vaddrs taddr* die Variablen-Knoten an den Adressen *vaddrs* durch die instanziierten Terme zu ersetzen. Etwas schwieriger ist es allerdings, wenn kein Sharing verwendet werden soll: In diesem Fall muss die Abstraktion durchlaufen, für jede Referenz auf eine dieser Variablen-Knoten der entsprechende Term kopiert und die Referenz ersetzt werden. Wir rufen an dieser Stelle einfach eine Funktion `tiInstantiateMetaApp1` auf und betrachten die Details des Verfahrens später.

Bei einer Kontext-Benutzung müssen wir ebenfalls, wie bei einer Meta-Applikation, ggf. gebundene Variablen umbenennen. Zunächst prüfen wir aber, ob es sich um eine Kontext-Benutzung der Form $C[X]$ handelt, wobei X eine Metavariable ist, die genau so auf der linken Seite der Regel vorgekommen ist, d.h. X ist genau die Metavariable, die wir zusammen mit den Informationen über den Kontext in der Assoziationsliste für Kontexte gespeichert haben. Ist dies der Fall, so liefern wir einfach `(heap, ctaddr)` zurück, wobei `heap` der unveränderte Heap ist und `ctaddr` die Adresse, die wir für den Kontext unserer Assoziationsliste entnehmen können. Handelt es sich jedoch nicht um diesen einfachen Sonderfall, so müssen wir, analog zur Meta-Applikation, den Term an der Adresse `ctaddr` kopieren, und erledigen mögliche Umbenennungen gebundener Variablen somit während des Kopiervorgangs. Allerdings benötigen wir hier eine spezielle Kopierfunktion `tiCopyCtxtTreeCBV`, die anhand der (ebenfalls in der Assoziationsliste gespeicherten) Datenstruktur vom Typ `IfrsCtxtTpl HeapAddress` die Lochposition bestimmt und für diese den instanziierten Lochterm einsetzt. Wir erzeugen dabei einen neuen Indirektions-Knoten, der auf den instanziierten Lochterm verweist – aufgrund des Sharings dürfen wir den alten Knoten im Loch nicht überschreiben, da sonst alle Unterterme, die sich diesen Knoten teilen, auf einmal ersetzt würden. Falls der Kontext auf der linken Seite in einer Abstraktion der Form $x.C[x]$ verwendet wurde und jetzt ebenfalls wieder in einer Abstraktion verwendet wird, die dieses x (evtl. umbenannt) enthält, so stellen wir zudem sicher, dass wir nun auf dieses neue x verweisen, damit diese alte Bindung erhalten bleibt. Dazu müssen wir unserer Kopier-Funktion zusätzlich ein Tupel übergeben, das den Namen der verwendeten Variablen im Term (können wir der Assoziationsliste für Kontexte entnehmen) und die Adresse der umbenannten Variablen (können wir der Assoziationsliste der gebunden Variablen entnehmen) enthält.

Bei einem Operatorterm instanziieren wir, analog zu `tiInstantiateAndUpdate1`, zunächst die Operanden anhand der Funktion `tiMapInstantiate`. Anschließend fügen wir einen neuen Applikations-Knoten in den Heap ein, der auf diese Operanden und die passende Operatordefinition verweist.

Als letzter Fall bleibt nun noch eine Domain übrig: Wir überprüfen hierbei, ob die Domain auf der linken Seite innerhalb einer Abstraktion verwendet wurde, d.h. ob in der entsprechenden Assoziationliste für diese Domain zusätzlich zu der Adresse `daddr` des zugehörigen Terms eine Liste der abstrahierten Variablen gespeichert wurde. Ist dies der Fall, so muss der Term an der Adresse `daddr` nämlich kopiert werden, falls nicht, können wir einfach `(heap, daddr)` zurückliefern (sofern Sharing verwendet wird). Beim Kopieren des Terms aus einer Abstraktion müssen wir, wie bei den Kontext-Benutzungen, darauf achten, dass die ursprünglichen Bindungen nicht durch Umbenennung bei der

Instanziierung einer Abstraktion aufgelöst werden. Wir benötigen also auch hier eine spezielle Kopier-Funktion `tiCopyDomTreeCBV`.

Wenden wir uns jetzt einer näheren Betrachtung der Instanziierungs-Funktionen `tiInstantiateAbs1` und `tiInstantiateMetaApp1` für Abstraktionen und Meta-Applikationen zu.

Instanziierung einer Abstraktion

Die Funktion `tiInstantiateAbs1` bekommt den zu instanziierenden Metaterm, den aktuellen Zustand und die Assoziationslisten mit den Informationen über benutzte Metavariablen, Kontexte, Domains und gebundene Variablen übergeben und soll die neue Abstraktion erstellen und deren Adresse zusammen dem geänderten Heap zurückliefern.

```
tiInstantiateAbs1 mt st mvars ctxts doms vars =
```

Die wichtigsten Berechnungen sind dabei Teil einer lokalen `let`-Definition. Zunächst extrahieren wir den Heap aus dem Zustand, außerdem die Variablen und den enthaltenen Metaterm aus der Abstraktion:

```
  let heap = tiHeap st
      avs = mtGetAbsVars mt st
      avnames = map (\x -> varName x st) avs
      aterm = mtGetAbsTerm mt st
```

Außerdem berechnen wir die Listen der im Metaterm `mt` enthaltenen Metavariablen, Kontexte und Domains:

```
      atmvars = map (\x -> mvarName x st) (mtMvars aterm st)
      atctxts = map (\x -> contextName x st)
                                          (mtCtxts aterm st)
      atdoms = map (\x -> domainName x st) (mtDoms aterm st)
```

Die Funktionen `mtMvars`, `mtCtxts`, `mtDoms` können dabei, analog zur Funktion `mtMvarsList` aus Abschnitt 6.3.2.2, leicht über die Struktur der Metaterme und mittels der Selektor-Methoden der Metaterm-Klasse implementiert werden. Die Funktionen `varName`, `mvarName`, `contextName` und `domainName` sind Selektor-Methoden der Variablen-, Metavariablen-, Kontext- und Domain-Klassen.

Als nächstes berechnen wir die freien Variablen aus den Instanzen der benutzten Metavariablen, Kontexte und Domains:

```
      ifreevars = tiInstGetFreeVars st (atmvars,mvars)
                                (atctxts,ctxts) (atdoms,doms)
```

Die Funktion `tiInstGetFreeVars` kann dabei leicht definiert werden, in dem wir für jedes Element aus den Listen `atmvars`, `atctxts` und `ctxt` die zugehörigen Adressen aus den Assoziationslisten `mvars`, `ctxts` und `doms` heraussuchen, und auf diese eine, analog zu `mtFreeVars1` aus Abschnitt 6.2.2.4 zu definierende Funktion

`trmFreeVarNames` anwenden (wir haben ja Adressen zu Instanzen der Term-Klasse gemacht) und die Ergebnisse konkatenieren. Bei Kontexten betrachten wir zwar unzulässigerweise den Lochterm im Argument mit, so dass wir evtl. freie Variablen aus diesem Lochterm in die Liste einfügen, die sonst nicht verwendet werden. Allerdings bedeutet das im Folgenden nur, dass wir evtl. auch Variablen umbenennen, bei denen dies nicht erforderlich wäre. Dies ist natürlich nicht schlimm, solange der Umbenennungsvorgang an sich richtig ist, d.h. die gewünschten Bindungen bleiben erhalten und es kommen keine neuen hinzu.

Neben der Liste der freien Variablen berechnen wir zudem eine Liste sämtlicher verwendeter Variablen:

```
ivars = tiInstGetVars st (atmvars,mvars)
                    (atctxts,ctxts) (atdoms,doms) vars
```

Die Funktion `tiInstGetVars` wird analog zu `tiInstGetFreeVars` definiert, außer dass wir anstelle der Funktion `trmFreeVarNames` die Funktion `trmVarNames` verwenden und zusätzlich noch die gebundenen Variablen aus der Assoziationsliste `vars` hinzufügen.

Es steht nun jeweils eine Liste der freien und überhaupt aller verwendeten Variablen in den Instanzen zur Verfügung. Wir haben allerdings noch nicht beachtet, dass der Metaterm `mt` an sich noch freie Variablen enthalten könnte, die Liste dieser Variablen berechnen wir auch noch:

```
mtfreevars = map (\x -> tiInstVarName
              (varName x st) vars) (mtFreeVars aterm st)
```

Die leicht zu definierende Funktion `tiInstVarName` schlägt dabei den neuen Namen der Variablen in der Assoziationsliste der gebundenen Variablen nach, falls diese durch eine äußere Abstraktion umbenannt wurde.

Jetzt können wir anhand einer Funktion `tiMaxIndex`, das maximale n errechnen, so dass x_n in der Liste der verwendeten Variablen enthalten ist, d.h. wir können sicher sein, dass es sich bei allen x_m, mit $m > n$, um neue, noch nicht verwendete Variablen handelt. Wir nennen dieses n hier `maxindex`:

```
maxindex = tiMaxIndex atvars
```

Dabei ist `atvars` die Vereinigung der Liste der verwendeten Variablen in den Instanzen mit der Liste der freien Variablen im Metaterm sowie der Liste der neuen abstrahierten Variablen in der zu instanziierenden Abstraktion:

```
atvars = mtfreevars ++ ivars ++ avnames
```

Nun können wir anhand einer Funktion `tiAbsReplaceVarNames` die abstrahierten Variablen umbenennen:

```
      ivs = tiAbsReplaceVarNames avnames ifreevars
                                              (maxindex + 1)
```

Jetzt können wir endlich die neuen Variablen-Knoten erzeugen:

```
     (heap1,addrs1) = heapInsertList heap
                               [(TiNVar n) | n <- ivs]
```

Den Metaterm `mt` instanziieren wir jetzt einfach durch einen rekursiven Aufruf von `tiInstantiate1`. Dabei müssen wir die neuen abstrahierten Variablen in die Assoziations-Liste der gebundenen Variablen einfügen, bevor wir diese übergeben. Da diese für jeden Variablennamen, wie er in der Regel vorkommt, ein Paar, bestehend aus dem (umbenannten) Namen der Instanz und der Adresse des instanziierten Variablen-Knotens, enthält, benötigen wir an dieser Stelle eine spezielle Funktion `tiVarsZip`, die analog zu `zip` definiert ist. Unser Aufruf gestaltet sich somit wie folgt:

```
      (heap2,addr2) = (tiInstantiate1 aterm
                        st{tiHeap = heap1} mvars ctxts doms
                          (assocInsertUpdateList vars
                            (tiVarsZip avnames ivs addrs1)))
```

Da wir bei der Instanziierung einer Variablen einfach nur die Adresse des bereits erstellten Variablen-Knotens in der Assoziationsliste der gebundenen Variablen suchen und zusammen mit dem unveränderten Heap zurückliefern, und da wir in dieser Assoziationsliste nun unter dem alten Namen den Namen und die Adresse der umbenannten Variable finden, bleiben also bei den Umbenennungen die gewünschten Bindungen erhalten. Abschließend können wir die Adressen der instanziierten Variablen und des instanzierten Terms in einen neuen Abstraktions-Knoten einsetzen:

```
         in
    heapInsert heap2 (TiNAbs addrs1 addr2)
```

Betrachten wir noch die Funktion `tiAbsReplaceVarNames`, die die Umbenennungen der Variablen vornimmt:

```
tiAbsReplaceVarNames [] freevars maxindex = []
tiAbsReplaceVarNames (vn:vns) freevars maxindex =
  if elem vn freevars then
    let nvn = ("x" ++ (show maxindex)) in
    (nvn:(tiAbsReplaceVarNames vns freevars (maxindex + 1)))
  else
    (vn:(tiAbsReplaceVarNames vns freevars maxindex))
```

Wie man sieht, wird für jedes Element der übergebenen Variablenliste geprüft, ob dieses in der Liste `freevars` der freien Variablen enthalten ist. Ist dies der Fall, so wird sie in `("x" ++ (show maxindex))` umbenannt, wobei wir `maxindex` bei jeder Umbenennung um Eins erhöhen.

Da wir durch unsere Berechnung von `maxindex` innerhalb der Funktion `tiInstantiateAbs1` sichergestellt haben, dass der neue Variablenname nicht anderweitig verwendet wird, können wir sicher sein, dass durch unsere Umbenennung keine neue unerwünschte Bindung entsteht. Die Funktion `tiMaxIndex`, die wir dabei verwendet haben, um `maxindex` aus der Liste aller verwendeten Variablen zu berechen, können wir einfach wie folgt definieren:

```
tiMaxIndex [] = 0
tiMaxIndex (vn:vnames) =
  let maxi = tiMaxIndex vnames in
    case vn of
      ('x':vs) ->
        case vs of
          [] -> maxi
          ds | all isDigit ds ->
            let maxn = read ds in
              max maxi maxn
          _ -> maxi
      _ -> maxi
```

Wir überprüfen also für jede Variable, ob deren Name aus einem „x“ gefolgt von einer Zahl besteht, falls ja merken wir uns diese Zahl, sofern diese bisher die größte verwendete Zahl war.

Damit sind unsere Betrachtungen zur Instanziierung einer Abstraktion abgeschlossen und wir wenden uns den Meta-Applikationen zu.

Instanziierung einer Meta-Applikation

Die Funktion `tiInstantiateAbs1` bekommt ebenfalls den zu instanziierenden Metaterm, den aktuellen Zustand und die vier Assoziationslisten übergeben und soll nun die Meta-Applikation durchführen, d.h. wir müssen den Term instanziieren, der sich durch die Substitution der Bindungsvariablen in der Metavariablen höherer Stelligkeit mit den Instanzen der eingesetzen Metaterme ergibt. Im Zusammenhang mit der Betrachtung der Funktion `tiInstantiate1` haben wir uns ja bereits ein paar Gedanken zum Vorgehen gemacht, deshalb widmen wir uns gleich der Haskell-Definition unserer Funktion:

```
tiInstantiateMetaApp1 mt st mvars ctxts doms vars =
```

Analog zur Funktion `tiInstantiateAbs1` sind die wichtigsten Berechnungen Teil einer lokalen `let`-Definition. Wir extrahieren zunächst die Metavariable und die eingesetzten Metaterme der Meta-Applikation:

```
  let mvar = mtGetMetaAppMVar mt st
      mvname = mvarName mvar st
      apterms = mtGetMetaAppTerms mt st
```

Anschließend schlagen wir die Adresse des zur Metavariablen zugehörigen Terms in der Assoziationsliste der Metavariablen nach:

```
mvaddr = head mvaddrs
(mvaddrs,(rvnames,tvnames)) =
   (assocLookup mvars mvname
    (error ("Undefined metavariable " ++
                                      mvname ++ "!")))
```

Analog zu `tiInstantiateAbs1` erstellen wir die Listen der in den Metatermen benutzten Metavariablen, Kontexte und Domains:

```
atmvars = (map (\x -> mvarName x st)
               (concatMap (\x -> mtMvars x st) apterms))
atctxts = (map (\x -> contextName x st)
               (concatMap (\x -> mtCtxts x st) apterms))

atdoms = (map (\x -> domainName x st)
              (concatMap (\x -> mtDoms x st) apterms))
```

Ebenso erstellen wir die Liste der freien Variablen innerhalb der Metaterme und innerhalb der verwendeten Instanzen der Metavariablen, Kontexte und Domains:

```
atmtfreevars =
   (map (\x -> tiInstVarName (varName x st) vars)
     (concatMap (\x -> mtFreeVars mt st) apterms))

atmvfreevars = tiInstGetFreeVars st (atmvars,mvars)
                           (atctxts,ctxts) (atdoms,doms)
```

Wir benötigen allerdings an dieser Stelle keinen Vergleich mit sämtlichen verwendeten Variablen, da wir nur bei enthaltenen Abstraktionen die in sich geschlossenen Bindungen umbenennen. Es reicht somit sicherzustellen, dass der neue Name nicht der Name einer freien Variablen ist:

```
maxindex = tiMaxIndex freevars
freevars = atmtfreevars ++ atmvfreevars
```

Nun rufen wir eine spezielle Kopier-Funktion auf, die beim Kopieren der Abstraktion die gebundenen Variablen umbenennt; diese liefert also einen neuen Heap und die Adresse der kopierten Abstraktion zurück:

```
(heap1,addr1) = (tiCopyTreeCBV mvaddr st freevars
                                       (maxindex + 1))
```

Damit sind alle vorab zu tätigenden Berechnungen abgeschlossen und wir können abschließend eine Funktion aufrufen, die in der kopierten Abstraktion die äußeren Bindungsvariablen durch die Instanzen der einzusetzenden Metaterme ersetzt und den neuen Heap, zusammen mit der Adresse des in der Abstraktion enthaltenen Terms, zurückliefert. Wie bereits erwähnt ist dabei ohne Sharing ein besonderes Vorgehen nötig, so dass wir hier entweder eine Funktion `tiUpdateAbsNode` oder eine Funktion `tiUpdateAbsNodeCV` aufrufen:

```
        in
    if (rsSharingOn () st) then
      tiUpdateAbsNode addr1 st{tiHeap=heap1} apterms
                                        mvars ctxts doms vars
    else
      tiUpdateAbsNodeCV addr1 st{tiHeap=heap1} apterms
                                        mvars ctxts doms vars
```

Da wir, wie gesehen, verschiedene Kopier-Funktion für Terme im Heap benötigen, wollen wir uns mit diesen, zusammen mit der Funktion `tiCopyTreeCBV`, später beschäftigen und betrachten zunächst die Funktion `tiUpdateAbsNode`:

```
tiUpdateAbsNode addr st terms mvars ctxts doms vars =
  let heap = tiHeap st
      node = heapLookup heap addr
        in
    case node of
      (TiNAbs addrs taddr) ->
        let st1 = tiUpdateAbsNode2 (zip addrs terms) st
                                        mvars ctxts doms vars
              in
          ((tiHeap st1),taddr)
      _ -> error "tiUpdateAbsNode: Unexpected heap node!"
```

Wir schlagen dabei einfach den Heap-Knoten an der übergebenen Adresse nach und stellen nochmals sicher, dass es sich um eine Abstraktion handelt. Anschließend bilden wir eine Liste `(zip addrs terms)` aus Paaren, bestehend aus jeweils einer Variablen-Adresse und dem einzusetzenden Term, und übergeben diese Liste der Funktion `tiUpdateAbsNode2`.

Diese definieren wir einfach wie folgt:

```
tiUpdateAbsNode2 [] st mvars ctxts doms vars = st
tiUpdateAbsNode2 ((addr,term):xs) st mvars ctxts doms vars =
  let st1 = tiInstantiateAndUpdate1 term st mvars
                                        ctxts doms vars addr
        in
    tiUpdateAbsNode2 xs st1 mvars ctxts doms vars
```

Wir überschreiben also jeden Variablen-Knoten mit dem instanziierten Term. Aufgrund des Sharings werden somit im Term alle (gewünschten) Variablen durch die entsprechenden Terme ersetzt.

Falls jedoch kein Sharing verwendet werden soll, so übergibt die Funktion `tiUpdateAbsNodeCV` die Adressen der Variablen und die einzusetzenden Terme nun an eine Funktion `tiUpdateAbsNodeCV2`, die den kopierten Abstraktions-Term durchläuft und alle Referenzen auf eine der übergebenen Variablen-Adressen durch Referenzen auf eigene Kopien der einzusetzenden Terme ersetzt, bzw. wir instanziieren die einzusetzenden Terme für jedes Vorkommen einer Variablen neu. Dadurch wird das

Sharing aufgehoben. Da das hierzu nötige Durchlaufen der Knoten des Abstraktions-Terms eine ähnliche Form hat wie bei den Kopiervorgängen, die wir jetzt im Anschluss betrachten, verzichten wir hier auf weitere Details und schließen somit unsere Betrachtungen zur Instanziierung einer Meta-Applikation ab.

Erstellung einer Kopie der Termdarstellung im Heap

Bei unserer Betrachtung der Term-Instanziierungen fehlen jetzt nur noch die Definitionen der Kopier-Funktionen für Terme. Wir wollen an dieser Stelle den einfachen Fall, einen Term ohne Umbenennung und ohne Sonderbehandlung für Kontexte oder Domains zu kopieren, detailliert betrachten und dann nur noch zusammenfassend auf die Unterschiede der anderen Kopier-Funktionen eingehen. Dieses einfache Kopieren übernimmt nun die Funktion `tiCopyTree`, welche die Adresse des zu kopierenden Terms und den Heap übergeben bekommt und einen neuen Heap zusammen mit der Adresse der Kopie zurückliefert:

```
tiCopyTree :: HeapAddress -> Heap TiNode ->
                                    (Heap TiNode,HeapAddress)
```

Wir wollen diese Funktion so gestalten, dass sie sowohl bei verwendetem Sharing als auch ohne Sharing eingesetzt werden kann. Daher müssen wir beim Kopieren aufpassen, dass wir nicht versehentlich das Sharing aufheben, indem wir Heap-Knoten, die von mehreren Untertermen geteilt werden, mehrfach kopieren und merken uns die Adressen der bereits kopierten Knoten. Wir verwenden dazu eine Assoziationsliste, in der wir für jede bereits kopierte Adresse die neue Adresse speichern. Somit rufen wir eine Unterfunktion `tiCopyTree1` auf, die wir mit der leeren Assoziationsliste initialisieren:

```
tiCopyTree addr heap =
  (heap1,addr1)
 where
  (heap1,addr1,_) = tiCopyTree1 addr heap assocEmpty
```

Diese Unterfunktion bekommt also die zu kopierende Adresse, den Heap und die Assoziationsliste der bereits kopierten Adressen übergeben und liefert neben dem neuen Heap und der neuen Adresse eine neue Assoziationsliste zurück:

```
tiCopyTree1 :: HeapAddress -> Heap TiNode ->
  Assoc HeapAddress HeapAddress ->
  (Heap TiNode,HeapAddress,Assoc HeapAddress HeapAddress)
```

Die Funktion überprüft, ob die übergebene Adresse bereits kopiert wurde, d.h. in der übergebenen Assoziationsliste enthalten ist. Falls ja, so geben wir einfach die in der Assoziationsliste gespeicherte neue Adresse zusammen mit dem unveränderten Heap und der unveränderten Assoziationsliste zurück. Falls nein, so kopieren wir den Unterterm mit Hilfe einer Funktion `tiCopyTree2`:

```
tiCopyTree1 addr heap asso =
  let laddr = (assocLookup asso addr (HeapAddress (-1))) in
    if ((heapAddress (laddr)) == (-1)) then
      let (heap1,addr1,asso1) = (tiCopyTree2 addr heap asso)
```

```
          in
       (heap1,addr1,(assocInsertF asso1 (addr,addr1)))
  else
    (heap,laddr,asso)
```

Die Funktion `tiCopyTree2` macht nun eine Fallunterscheidung über den an der übergebenen Adresse gespeicherten Heap-Knoten:

```
tiCopyTree2 addr heap asso =
  case heapLookup heap addr of
```

Bei einem Applikations-Knoten kopieren wir anhand einer Funktion `tiCopyTrees` die Operanden und fügen dann einen neuen Applikations-Knoten mit Verweisen auf die kopierten Operanden in den Heap ein:

```
    TiNAp opname opaddr taddrs ->
      let (heap2,addr2) = heapInsert heap1
                            (TiNAp opname opaddr addrs1)
          (heap1,addrs1,asso1) = tiCopyTrees taddrs heap asso
            in
          (heap2,addr2,asso1)
```

Ein `TiNRule`-Knoten sollte nicht vorkommen, da dieser kein Term ist:

```
    TiNRule opname opdef ->
      error "Rule does not need to be copied!"
```

Ein `TinCAF`-Knoten braucht nicht kopiert zu werden:

```
    TiNCAF opname opdef mbaddr ->
      (heap,addr,asso)
```

Bei einem Abstraktions-Knoten kopieren wir zunächst die Variablen anhand einer Funktion `tiCopyTVars`, anschließend rufen wir rekursiv `tiCopyTree1` auf, um den in der Abstraktion enthaltenen Term zu kopieren. Da wir die Adressen der kopierten Variablen in die Assoziationsliste einfügen, bzw. `tiCopyTVars` eine entsprechende neue Assoziationsliste zurückliefert, werden Verweise auf die abstrahierten Variablen beim rekursiven Aufruf von `tiCopyTree1` richtig, d.h. auf die Adressen der Kopien, gesetzt. Anschließend fügen wir einen neuen Abstraktions-Knoten mit Verweisen auf die Kopien in den Heap ein:

```
    TiNAbs vaddrs taddr ->
      let (heap3,addr3) = heapInsert heap2
                            (TiNAbs addrs1 addr2)
          (heap1,addrs1,asso1) = tiCopyTVars vaddrs heap asso
          (heap2,addr2,asso2) = tiCopyTree1 taddr heap1 asso1
            in
         (heap3,addr3,asso2)
```

Bei einer Indirektion folgen wir dem Verweis:

```
    TiNInd iaddr -> tiCopyTree1 iaddr heap asso
```

Falls wir auf eine Variable treffen, handelt es sich um eine lokal (d.h. innerhalb des kopierten Terms) freie Variable, da wir Variablen von einem Abstraktions-Knoten ausgehend mit der Funktion `tiCopyTVars` kopieren. Wir können aber nicht einfach einen neuen Variablen-Knoten erzeugen, da diese Variable vermutlich durch eine äußere Abstraktion, in welcher der kopierte Term ein Unterterm ist, gebunden ist (da wir sichergestellt haben, dass die Reduktionsregeln des Reduktionssystems keine freien Variablen erzeugen). Somit erstellen wir eine Indirektion auf den alten Knoten:

```
    TiNVar vname ->
        let (heap1,addr1) = heapInsert heap (TiNInd addr) in
            (heap1,addr1,asso)
```

Damit sind alle Fälle abgedeckt. Die Funktion `tiCopyTrees` ruft nun für jedes Element der Liste `tiCopyTree1` auf, wobei wir darauf achten müssen, stets den neuen Heap und die neue Assoziationsliste weiterzureichen:

```
tiCopyTrees [] heap asso = (heap,[],asso)
tiCopyTrees (addr:addrs) heap asso =
  let (heap1,addr1,asso1) = tiCopyTree1 addr heap asso
      (heap2,addrs2,asso2) = tiCopyTrees addrs heap1 asso1
        in
    (heap2,(addr1:addrs2),asso2)
```

Ähnlich verfährt die Funktion `tiCopyTVars`, nur dass diese statt `tiCopyTree1` eine Funktion `tiCopyTVar` aufruft, die einen Variablen-Knoten kopiert:

```
tiCopyTVars [] heap asso = (heap,[],asso)
tiCopyTVars (addr:addrs) heap asso =
  let (heap1,addr1,asso1) = tiCopyTVar addr heap asso
      (heap2,addrs2,asso2) = tiCopyTVars addrs heap1 asso1
        in
    (heap2,(addr1:addrs2),asso2)

tiCopyTVar addr heap asso =
  case heapLookup heap addr of
    TiNVar vname ->
      let (heap1,addr1) = heapInsert heap (TiNVar vname) in
        (heap1,addr1,(assocInsertF asso (addr,addr1)))
```

Somit können wir einen Term kopieren. Die Funktion `tiCopyTree` kann sowohl bei verwendetem Sharing als auch ohne Sharing eingesetzt werden, denn es teilen sich in der Kopie nur die Unterterme gemeinsame Knoten, die sich auch im ursprünglichen Term Knoten teilen.

Der Kopiervorgang lässt sich leicht modifizieren, um dabei gleichzeitig Umbenennungen der gebundenen Variablen vornehmen zu können. Die Funktion `tiCopyTreeCBV` bekommt dazu zusätzlich die Liste der freien Variablen und den Index, der in den neuen

Variablennamen verwendet werden soll, übergeben. Anstelle der Funktion `tiCopyTVar` wird dann eine Funktion `tiCopyVarCBV` verwendet, um die Variablen bei enthaltenen Abstraktionen zu kopieren. Diese prüft nach, ob eine zu kopierende Variable in der Liste der freien Variablen enthalten ist, falls ja wird diese umbenannt in x_n, wobei n der übergebene Index ist und dieser bei jeder Umbenennung um Eins erhöht wird.

Bei einem Kopiervorgang für die Instanziierung einer Kontext-Benutzung wird zudem die Datenstruktur `IfrsCtxtTpl` aus dem Kontext-Matching übergeben. Diese hat bzgl. Operatoren und Operanden die gleiche Struktur wie der übergebene Term (da diese ja während des Kontext-Matchings extra so konstruiert wurde). Somit brauchen wir bei der Fallunterscheidung nur diese Datenstruktur mitzubetrachten, um so erkennen zu können, welcher Unterterm der Lochterm ist. Für diesen Lochterm legen wir als Platzhalter einen neuen Indirektionsknoten an und liefern zusammen mit der Adresse des gesamten kopierten Terms die Adresse dieses Indirektionsknotens als Loch-Adresse zurück. Bei der Instanziierung der Kontext-Benutzung brauchen wir somit nur noch den Indirektions-Knoten an der Loch-Adresse mit einer Indirektion auf den instanziierten Metaterm, der in das Loch eingesetzt werden soll, zu überschreiben.

Um bei Kontexten und Domains, die auf der linken Seite der Regel innerhalb einer Abstraktion der Form $x.C[x]$ bzw. $x_1 \ldots x_n.D$ eingeführt wurden und die auf der rechten Seite innerhalb einer Abstraktion mit den gleichen Bindungsvariablen verwendet werden, bei Umbenennungen die ursprünglichen Bindungen wiederherstellen zu können, reicht es, wie bei der Betrachtung der Funktion `tiInstantiate1` geschildert, eine Assoziationsliste mit den alten und neuen Variablennamen sowie der Adresse der zugehörigen Variablenknoten an die Kopierfunktion zu übergeben. Während des Kopiervorgangs kann dann bei den entsprechenden Variablen die benötigte Adresse des Variablenknotens nachgeschlagen werden, somit wird sichergestellt, dass die gewünschten Bindungen erhalten bleiben.

Somit haben wir alle benötigten Funktionen zum Anwenden einer Regel betrachtet.

6.2.4.3 Ausgabe der Ergebnisse

Als Ergebnis unserer Auswertungs-Funktion `tiEval` haben wir jetzt zunächst eine Liste von Zuständen der Template Instantiation Machine, die zudem noch durch den Fehler-Typ `TiResult` gekapselt sind. Wir benötigen also noch eine Funktion `irsShow`, welche die Ergebnis-Liste in eine sinnvolle Ausgabe umwandelt. Für jedes Element dieser Liste prüfen wir dabei anhand der Funktion `errResultOk`, ob der Reduktionsschritt erfolgreich ausgeführt werden konnte. Ist dies der Fall, so schlagen wir die obere Adresse im Stack nach, der ja Teil des Zustands ist. Diese Adresse enthält den reduzierten Term; da wir Adressen direkt als Terme verwenden können, wandeln wir nun mit Hilfe einer Funktion `termStr` den Term in eine String-Darstellung um. Die Funktion `termStr` lässt sich dabei leicht über die Struktur der Terme definieren, indem die Methoden der Term-Klasse verwendet werden. Falls der Reduktionsschritt jedoch nicht erfolgreich ausgeführt werden konnte, so wandeln wir anhand der Funktion `errGetMsg` die Fehler-Information in einen String um, der die Fehlerbeschreibung enthält. Wir haben somit eine Liste von Strings und können nun der Reihe nach die Elemente dieser Liste mittels der IO-Methode `putStrLn` ausgeben. Abschließend können wir eine Interpreter-Funktion

`interpret` definieren, welche ein Reduktionssystem übergeben bekommt und darauf basierend die Reduktion ausführt und die Ergebnisse ausgibt:

```
interpret :: IfrsRedSys -> IO ()
interpret rs = irsShow (tiEval (tiStart rs))
```

6.2.5 Start des Interpreters

Nachdem wir nun alle benötigten Bestandteile unseres Softwaresystems genauer betrachtet haben, fehlt als letztes noch eine Funktion, die diese Bestandteile verbindet, also die Lade-Prozedur für eine gewünschte Eingabedatei in Gang setzt, auf das eingelesene Reduktionssystem die Analyse-Funktion anwendet und schließlich die Reduktion veranlasst. Diese können wir leicht wie folgt definieren:

```
runInterpreter :: String -> IO ()
runInterpreter file =
  do
  rs <- loadRS file
  analyse rs
  interpret rs
```

Somit können wir nun z.B. im Interpreter Hugs die Reduktion für eine Datei *beispiel.txt* durch folgenden Aufruf veranlassen:

```
runInterpreter "beispiel.txt"
```

Um unser Programm aber auch selbstständig ausführbar nutzen zu können, müssen wir noch im Modul `Main` eine `main`-Funktion definieren. Wir lassen uns dabei durch die Haskell-Funktion `getArgs` die Kommandozeilen-Argumente zurückliefern und rufen dann eine Funktion `processArgs` auf, die diese Argumente verarbeitet:

```
main :: IO ()
main =
  do
    args <- getArgs
    processArgs args
```

Der Einfachheit halber rufen wir in der Funktion `processArgs` bei genau einem Argument die Funktion `runInterpreter` auf, die dann den übergebenen Parameter als Dateinamen behandelt und versucht, die entsprechende Datei zu laden und zu interpretieren. Ansonsten rufen wir eine Funktion `printHelp` auf, in der wir dann einen kurzen Text ausgeben, der über den richtigen Umgang mit dem Programm informiert:

```
processArgs :: [String] -> IO ()
processArgs [] = printHelp
processArgs [arg] = runInterpreter arg
processArgs (arg:args) = printHelp
```

Wir können nun unser Programm in Maschinensprache kompilieren lassen, z.B. durch den GHC mittels folgendem Aufruf:

```
ghc --make main.hs
```

Somit haben wir ein ausführbares Programm, wobei wir nun den Interpreter mit folgendem Kommandozeilen-Aufruf starten können, sofern wir die von GHC erstellte Datei entsprechend umbenennen:

`ifrs` *Dateiname*

6.3 Ergebnisse

Um unser Programm testen zu können, benötigen wir geeignete Eingabedateien. Wir wollen zunächst einige ausgewählte Beispiele betrachten, die die Arbeitsweise des Programms verdeutlichen. Anschließend untersuchen wir anhand einer etwas aufwendigeren Eingabe die Laufzeit des Programms.

6.3.1 „Lazy“ Lambda-Kalkül mit Sharing

Anhand unserer Vorüberlegungen in Abschnitt 5.4.1 können wir unser Reduktionssystem durch leichte Anpassung der Syntax wie folgt definieren:

```
\(x.X{x}) = constructor

@(\(x.X{x}),Y)=X{Y}
```

Wir stellen dabei den Operator λ durch das Zeichen \ dar.

Als einfaches Beispiel definieren wir folgenden `main`-Term:

```
main() = @(\(x.@(x,x)),@(\(x.x),\(x.x)))
```

Gemäß unseren Vereinbarungen zur Syntax der Reduktionssystems-Definitionen wird Sharing standardmäßig verwendet. Wir können dies aber auch explizit fordern, indem wir vor den eigentlichen Regeln den Befehl

```
sharing = on
```

einfügen.

Starten wir nun den Interpreter mit dieser Definition, so ergibt sich folgende Auswertung:

```
main()

@(\(x.@(x,x)),@(\(x.x),\(x.x)))

@(@(\(x.x),\(x.x)),@(\(x.x),\(x.x)))

@(\(x.x),\(x.x))

\(x.x)
```

Wie wir sehen, verläuft die Reduktion wie gewünscht und der Term `@(\(x.x),\(x.x))`, der ja durch Einsetzen in die Lambda-Funktion `\(x.@(x,x))` verdoppelt wird, wird nur einmal zu `\(x.x)` ausgewertet.

6.3.2 „Lazy“ Lambda Kalkül ohne Sharing

Den Unterschied zwischen der Verwendung von Sharing und der Auswertung ohne Sharing sehen wir, wenn wir die Sharing-Einstellung in

```
sharing = off
```

ändern.

Die Auswertung wird nun wie folgt durchgeführt:

```
main()

@(\(x.@(x,x)),@(\(x.x),\(x.x)))

@(@(\(x.x),\(x.x)),@(\(x.x),\(x.x)))

@(\(x.x),@(\(x.x),\(x.x)))

@(\(x.x),\(x.x))

\(x.x)
```

Wie wir sehen, wird ein Reduktionsschritt mehr benötigt, da der kopierte Term jetzt ebenfalls ausgewertet werden muss.

6.3.3 Call-by-Value Lambda Kalkül (ohne Sharing)

Modifizieren wir das Beispiel noch so, dass beide Operanden für @ strikt sind. Wir erhalten somit den Call-by-Value Lambda Kalkül:

```
sharing = off

\(x.X{x}) = constructor

strict(@) = 1,2
@(\(x.X{x}),Y)=X{Y}

main() = @(\(x.@(x,x)),@(\(x.x),\(x.x)))
```

Die Auswertung lautet nun:

```
main()

@(\(x.@(x,x)),@(\(x.x),\(x.x)))
```

```
@(\(x.@(x,x)),\(x.x))

@(\(x.x),\(x.x))

\(x.x)
```

Wie man sieht, muss im Term `@(\(x.@(x,x)),@(\(x.x),\(x.x)))` zuerst der zweite Operand `@(\(x.x),\(x.x))` ausgewertet werden (der erste ist ja bereits ein Wert), bevor die Regel für `@` auf den gesamten Term angewendet werden kann.

6.3.4 PCF mit Peano-Zahlen

Wir definieren hier eine Variante von PCF, in der wir die natürlichen Zahlen durch Peano-Zahlen ausdrücken. Eine Peano-Zahl ist dabei Null oder der Nachfolger einer Peano-Zahl, wir benutzen hier die beiden folgenden Konstruktoren:

```
0()=constructor

S(X)=constructor
```

Aufgrund des fehlenden Typsystems können wir allerdings nicht sicherstellen, dass für `S(X)` das `X` eine Peano-Zahl enthält. Dies kann bei falscher Verwendung also zu Laufzeitfehlern führen. Die Definition der restlichen Operatoren folgt nun unseren Überlegungen aus Abschnitt 5.4.4, mit kleinen Anpassungen an die Syntax und die Verwendung der Peano-Zahlen:

```
\(x.X{x}) = constructor

@(\(x.X{x}),Y)=X{Y}

true()  = constructor
false() = constructor

ifthenelse(true(),X,Y) = X
ifthenelse(false(),X,Y)= Y

pred(0()) =0()
pred(S(X))=X

succ(X) = S(X)

zero?(0()) = true()
zero?(X)  = false()

μ(x.X{x})=X{μ(x.X{x})}
```

Um ein interessanteres Beispiel zu erhalten, definieren wir noch eine Addier-Funktion

```
add()=μ(a.\(s.\(t.ifthenelse(zero?(s),t,
                                  succ(@(@(a,pred(s)),t))))))
```

und betrachten folgenden Term:

```
main()=@(@(add(),S(0())),S(0()))
```

Die Reduktion verläuft jetzt wie folgt:

```
main()

@(@(add(),S(0())),S(0()))

@(@(μ(a.\(s.\(t.ifthenelse(zero?(s),t,succ(@(@(a,pred(s)),t))
 ))))),S(0())),S(0()))

@(@(\(s.\(t.ifthenelse(zero?(s),t,succ(@(@(μ(x.\(s.\(
 t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),t))))))),
 pred(s)),t))))),S(0())),S(0()))

@(\(t.ifthenelse(zero?(S(0())),t,succ(@(@(μ(x.\(s.\(
 t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),t))))))),
 pred(S(0()))),t)))),S(0()))

ifthenelse(zero?(S(0())),S(0()),succ(@(@(μ(x.\(s.\(
 t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),t))))))),
 pred(S(0()))),S(0()))))

ifthenelse(false(),S(0()),succ(@(@(μ(x.\(s.\(t.ifthenelse(
 zero?(s),t,succ(@(@(x,pred(s)),t))))))),pred(S(0()))),
 S(0()))))

succ(@(@(μ(x.\(s.\(t.ifthenelse(zero?(s),t,succ(@(@(x,
 pred(s)),t))))))),pred(S(0()))),S(0())))

S(@(@(μ(x.\(s.\(t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),
 t))))))),pred(S(0()))),S(0())))
```

Wie man sieht, wird nur bis zum (äußeren) Auftreten des Konstruktors S ausgewertet, der Operand wird also, wie bei lazy-Auswertung üblich, nicht ausgewertet. Dies können wir aber leicht ändern, in dem wir den Konstruktor S strikt machen. Wir ändern die Definition für S also um in:

```
strict(S) = 1
S(X)=constructor
```

So wird die Auswertung beim Auftreten von S fortgesetzt, bis auch der Operand ein Wert ist:

```
S(@(@(\(s.\(t.ifthenelse(zero?(s),t,succ(@(@(μ(x.\(s.\(
 t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),t))))))),
 pred(s)),t))))),pred(S(0()))),S(0())))
```

```
S(@(\(t.ifthenelse(zero?(pred(S(0()))),t,succ(@(@(μ(x.\(s.\(
 t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),t)))))),pred(
 pred(S(0()))),t)))),S(0())))

S(ifthenelse(zero?(pred(S(0()))),S(0()),succ(@(@(μ(x.\(s.\(
 t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),t)))))),pred(
 pred(S(0()))),S(0()))))))

S(ifthenelse(zero?(0()),S(0()),succ(@(@(μ(x.\(s.\(
 t.ifthenelse(zero?(s),t,succ(@(@(x,pred(s)),t)))))),
 pred(0())),S(0()))))))

S(ifthenelse(true(),S(0()),succ(@(@(μ(x.\(s.\(t.ifthenelse(
 zero?(s),t,succ(@(@(x,pred(s)),t)))))),pred(0())),S(0()))))))

S(S(0()))
```

Da unser Term `main()=@(@(add(),S(0())),S(0()))` der Berechnung 1 + 1 entspricht, kommt mit `S(S(0()))`, was der Zahl 2 entspricht, das richtige Ergebnis heraus.

Um anstelle der Peano-Zahlen Integer-Zahlen zu verwenden, könnte man unser System analog zu [PJL91, Abschnitt 2.6] so erweitern, dass wir ausgewählte Haskell-Datentypen als primitive Datentypen benutzen können.

6.3.5 Call-by-Need Lambda Kalkül

Unser Ziel war es ja, neben den GDSOS-kompatiblen Reduktionssystemen, auch den Call-by-Need Lambda Kalkül darstellen zu können. Dazu mussten wir schließlich mit der Betrachtung von (Reduktions-) Kontexten und Domains auch einen erheblichen Mehraufwand betreiben. Daher wollen wir auch untersuchen, wie die Darstellung dieses Kalküls in unserem System aussieht und wie sich die Reduktion verhält:

Die Definition des Reduktionskontextes (Definition 4.3.4.) lässt sich leicht an unsere Syntax anpassen:

```
reductioncontext R = [] | @(R,t) | let(x.R,t)
                   | let (x.R[x],R)
```

Antworten können wir nun als Werte-Domain definieren:

```
value domain A = \(x.t) | let(x.A,t)
```

Natürlich brauchen wir auch wieder die Definition des Lambda-Operators:

```
\(x.X{x}) = constructor
```

Um die Verwendung der Lambda-Abstraktionen als kopierbare Ausdrücke bzw. Werte zu verdeutlichen, definieren wir auch noch eine Domain `V`:

```
domain V = \(x.t)
```

Wir benötigen an dieser Stelle das Schlüsselwort `value` nicht, da wir ja bereits `\` als kanonischen Operator definiert haben.

Somit können wir die Reduktionsregeln aus Definition 4.3.5. leicht darstellen:

```
@(\(x.X{x}),Y) = let(x.X{x},Y)    -- (Ino)
@(let(x.A,L),N) = let(x.@(A,N),L) -- (Cno)

let(x.R[x],V) = let(x.R[V],V)                    -- (Vno)
let(y.R[y],let(x.A,L)) = let(x.let(y.R[y],A),L) -- (Ano)
```

Als auszuwertenden Term betrachten wir:

```
main() = @(\(x.@(x,x)),@(\(y.y),\(z.z)))
```

Führen wir nun diese Definition mit unserem Interpreter aus, so erhalten wir folgendes Ergebnis:

```
Warnung: "beispiel let.txt":19,3: GDSOS-Bedingung 1 nicht
erfüllt: Strikte Positionen müssen einfache Meta-Werte sein!

Warnung: Reduktionskontext wurde modifiziert!

main()

@(\(x.@(x,x)),@(\(y.y),\(z.z)))

let(x.@(x,x),@(\(y.y),\(z.z)))

let(x.@(x,x),let(x.x,\(z.z)))

let(x.@(x,x),let(x1.\(z.z),\(z.z)))

let(x1.let(y.@(y,y),\(z.z)),\(z.z))

let(x1.let(x.@(\(z.z),x),\(z.z)),\(z.z))

let(x1.let(x.let(x.x,x),\(z.z)),\(z.z))

let(x1.let(x1.let(x.\(z.z),x1),\(z.z)),\(z.z))
```

Unsere Analyse-Funktionen geben zunächst Warnungen aus, dass die erste GDSOS-Bedingung nicht erfüllt ist und die Definition des Reduktionskontextes vom Benutzer verändert wurde. Auch die Reduktion des Terms sieht auf den ersten Blick etwas ungewohnt aus, da ausgerechnet das Ergebnis der längste Term ist. Aber wie man sieht, werden die Regeln richtig angewendet und das Ergebnis ist eine Antwort.

6.3.6 Laufzeitverhalten am Beispiel eines aufwendigeren KFP-Programms

Als Prüfstein für die Leistung unseres System wollen wir nun die Darstellung des Quicksort-Algorithmusses in KFP aus [Sch00, Abschnitt 3.1] betrachten.

Wir benötigen zunächst die üblichen Regeln des Lambda-Kalküls:

```
\(x.X{x}) = constructor

@(\(x.X{x}),Y) = X{Y}
```

Dann definieren wir die Bool'schen Werte, sowie ein $\texttt{Case}_{\texttt{Bool}}$, dass wir hier `IF` nennen:

```
True() = constructor
False()= constructor

IF(True(),x,y) = x
IF(False(),x,y)= y
```

Außerdem benötigen wir eine Datenstruktur für Paare:

```
Paar(x,y)=constructor

fst(x,y)=x
snd(x,y)=y

CASEPAAR(Paar(X,Y),x y.A{x,y})=A{X,Y}
```

Listen dürfen natürlich auch nicht fehlen. Wir definieren an dieser Stelle die zweite Position von `Cons` als strikt, damit die Liste stets vollständig (der Länge nach) ausgewertet wird:

```
Nil()=constructor
strict(Cons)=2
Cons(x,y)=constructor

CASELIST(Nil(),Y1,y ys.Y2{y,ys})=Y1
CASELIST(Cons(X,Y),Y1,y ys.Y2{y,ys})=Y2{X,Y}
```

Insbesondere benötigen wir einen Append-Operator `++`:

```
++() = \(xs.\(ys.CASELIST(xs,ys,h t.Cons(h,@(@(++(),t),
                                                ys)))))
```

Wir benutzen auch hier Peano-Zahlen, wobei wir aber nun zusätzlich ein spezielles `case` benötigen:

```
0()=constructor
strict(S)=1
S(x)=constructor
```

```
CASEINT(0(),X,y.Y{y})=X
CASEINT(S(Z),X,y.Y{y})=Y{Z}
```

Außerdem benötigen wir Vergleichsoperatoren < und ≤, die wir hier `lt` und `leq` nennen:

```
lt() = \(x.\(y.CASEINT(x,CASEINT(y,False(),z.True()),
                xx.CASEINT(y,False(),yy.@(@(leq(),xx),yy)))))

leq() = \(x.\(y.CASEINT(x,True(),xx.CASEINT(y,False(),
                                    yy.@(@(leq(),xx),yy)))))
```

Somit können wir den Quicksort-Algorithmus wie folgt darstellen:

```
partition() = \(p.\(xs.CASELIST(xs,Paar(Nil(),Nil()),
 h t.CASEPAAR(@(@(partition(),p),t),
  oks noks.IF(@(p,h),Paar(Cons(h,oks),noks),
   Paar(oks,Cons(h,noks)))))))

quicksort() = \(xs.CASELIST(xs,Nil(),
 h t.CASEPAAR(@(@(partition(),@(lt(),h)),t),
  kleine grosse.@(@(++(),@(quicksort(),kleine)),
   Cons(h,@(quicksort(),grosse))))))
```

Die Funktion `quicksort` sortiert dabei eine Liste von Peano-Zahlen in absteigender Reihenfolge.

Als Beispielliste definieren wir

```
bsp_list() = Cons(0(),Cons(1(),Cons(2(),Cons(3(),Nil())))),
```

wobei wir folgende Abkürzungen für Peano-Zahlen definieren:

```
1() = S(0())
2() = S(S(0()))
3() = S(S(S(0())))
```

Als `main`-Term definieren wir somit die Anwendung der `quicksort`-Funktion auf unsere Beispielliste:

```
main() = @(quicksort(),bsp_list())
```

Ergebnis

Unser Interpreter reduziert diesen Ausdruck nun in 176 Schritten[43] zu

[43] Die Anzahl der Reduktionsschritte kann man sich leicht durch eine kleine Modifikation unserer Ausgabe-Funktion anzeigen lassen. Das Erstellen des initialen Zustands wird dabei ebenfalls als Schritt gezählt.

```
Cons(S(S(S(0()))),Cons(S(S(0())),Cons(S(0()),Cons(0(),
                                                  Nil()))))
```

d.h. unsere Liste `[0,1,2,3]` wurde korrekt absteigend sortiert zu `[3,2,1,0]`.

Auf einem handelsüblichen PC (750 MHz, 256 MB RAM) kann der mit dem GHC Version 5.04.2 kompilierte Interpreter diese 176 Reduktionsschritte inkl. Laden der Datei und Überprüfung der GDSOS-Bedingungen in ca. 14 Sekunden ausführen. Dies ist im Hinblick darauf, dass wir bisher keinerlei Maßnahmen ergriffen haben, um die Ausführungsgeschwindigkeit unseres Programms zu optimieren, ein sehr zufriedenstellender Wert. Insbesondere muss hier berücksichtigt werden, dass der Redex stets über einen Vergleich mit dem (implizit berechneten) Reduktionskontext gefunden wurde. Für GDSOS-kompatible Reduktionssysteme könnte somit durch eine Optimierung der Redexsuche, wie in Abschnitt 6.2.4 angedeutet, eine erhebliche Beschleunigung erreicht werden. Führen wir unseren Quicksort-Algorithmus allerdings im Zusammenhang mit einer etwas längeren Beispielliste aus, so stellen wir fest, dass mit zunehmender Dauer die einzelnen Reduktionsschritte immer mehr Zeit benötigen. Dies liegt daran, dass sich durch die Verwendung von Indirektions-Knoten die Termdarstellungen immer mehr aufblähen, ebenso wird reichlich Speicherplatz im Heap verschwendet, da wir nicht mehr benötigte Knoten nicht löschen. Abhilfe kann hier eine *Garbage Collection*, wie in [PJL91, Abschnitt 2.9] beschrieben, schaffen, welche die nicht mehr verwendeten Knoten löscht und ebenso Indirektionen durch Herstellung einer direkten Verknüpfung entfernen kann.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Wir haben ein Softwaresystem entwickelt, das in der Lage ist, Beschreibungen von Termersetzungssystemen höherer Ordnung, deren Reduktionsregeln auf einer strukturellen operationalen Semantik basieren, einzulesen und zu interpretieren. Das System ist dabei fähig, Reduktionskontexte für die Redexsuche zu benutzen, die entweder vom Benutzer definiert werden können oder automatisch anhand der strikten Positionen berechnet werden. Außerdem dürfen Kontexte und spezielle Definitionen für Term-Mengen, die wir Domains nennen, in den Reduktionsregeln verwendet werden.

Mit dem resultierenden Reduktionssystem-Format können wir somit nicht nur den „lazy" Lambda-Kalkül, den Call-by-Value Lambda-Kalkül und verwandte, um Konstruktoren und Fallunterscheidungen erweiterte Kalküle, wie die in Kapitel 4 vorgestellten Kernsprachen KFP und PCF, darstellen, sondern auch den (in Abschnitt 4.3 vorgestellten) Call-by-Need Lambda-Kalkül, welcher sich durch die Verwendung von Kontexten innerhalb der Regeln deutlich von den anderen Kalkülen abhebt.

Allerdings hält sich der Call-by-Need Lambda-Kalkül damit nicht an das in Kapitel 5 vorgestellte GDSOS-Format, das u.a. sicherstellt, dass Bisimulation eine Kongruenz ist. Wir haben dabei in Abschnitt 5.3.3 bewiesen, dass sich ein GDSOS-Reduktionssystem in ein äquivalentes strukturiertes Auswertungssystem nach Howe übersetzen lässt. Unser System ist in der Lage, die GDSOS-Bedingungen zu prüfen und gibt eine Warnung aus, falls eine der nötigen Bedingungen nicht erfüllt ist (wobei aus dieser auch gleich der Grund des Verstoßes hervorgeht).

Wie wir gesehen haben, ist unser System nicht nur befähigt, die einzelnen Reduktionsschritte für kleinere Bespiele ordnungsgemäß auszuführen, sondern es ist durchaus in der Lage, auch aufwendigere KFP-Ausdrücke, wie in unserem Quicksort-Beispiel, auszuwerten.

7.2 Ausblick

Da wir bei unserer Implementierung zunächst einmal nicht so sehr auf den Resourcenverbrauch bzgl. Ausführungsgeschwindigkeit und Speicherplatz geachtet haben, ist hier jedoch noch reichlich Spielraum für Optimierungen. Wie bereits in Abschnitt 6.3.6 beschrieben, wäre da zum einen die Möglichkeit, eine spezielle Behandlung für GDSOS-kompatible Reduktionssysteme zu realisieren, die bei der Redexsuche nicht den Reduktionskontext, sondern direkt die strikten Operanden überprüft. Zum anderen könnte eine Garbage Collection den Speicherbedarf erheblich reduzieren und auch die Ausführungsgeschwindigkeit verbessern. Außerdem gibt es natürlich noch jede Menge weiterer Optimierungsmöglichkeiten, u.a. wären hier bessere Suchstrukturen (z.B. durch eine Hash-Tabelle) für Operatordefinitionen zu nennen.

Neben dem Resourcenverbrauch wäre auch das Design einer ausdruckskräftigen Schnittstelle für andere Programme ein lohnendes Ziel für Erweiterungen. Bisher stehen

dazu ausschließlich unsere Unterfunktionen zur Verfügung, die bei einer Einbindung unserer Module benutzt werden können. Zwar haben wir diese speziell für diese Einsatzmöglichkeit allgemein gehalten, indem wir z.B. auch Nichtdeterminismus mitbetrachten und bei der Regelsuche gleich die Liste aller passenden Regeln zurückliefern. Besonders elegant wäre es jedoch, wenn man, wie bei monadischen Parser-Kombinatoren, eine einfache Kombinationsmöglichkeit dieser Funktionen hätte, um so z.B. leichter beschreiben zu können, welche Möglichkeiten bei einer nichtdeterministischen Auswertung weiterverfolgt werden sollen.

Aber auch mit den zur Verfügung stehenden Modulen lohnt es sich, unser System überall dort einzusetzen, wo Reduktionen benötigt werden. Es wäre also eine gute Basis für Programmanalysen, die mit Reduktionen arbeiten, wie z.B. eine Striktheitsanalyse oder die bereits in den Vorüberlegungen (Abschnitt 6.1) erwähnte Gleichheitsanalyse.

Neben möglichen Erweiterungen für unser Softwaresystem oder dessen Einsatz in anderen Programmen ist auch eine weitere Grundlagenforschung sinnvoll. So wäre es ein großer Fortschritt, wenn man die Eigenschaften für GDSOS-kompatible Reduktionssysteme auch für erweiterte Systeme, deren Regeln Kontexte und Domains enthalten dürfen, zeigen könnte.

Anhang A: Abfangen von Fehlern

Um Fehler abfangen zu können, kapseln wir die Berechnung durch eine Fehler-Monade. Das Grundprinzip ist in [MG01, Abschnitt 2.5.1.] gut dargestellt.

A.1 Die vordefinierte Klasse MonadError

Bei der Benutzung von Haskell-Erweiterungen, insbesondere Multiparameter-Klassen, steht im Hugs und im GHC bereits eine besondere Klasse `MonadError` zur Verfügung. Leider sind bei der Verwendung des Hugs-Interpreters andere `import`-Deklarationen nötig als beim Compiler GHC.

Für den Hugs-Interpreter genügt:

```
import MonadError
```

Der Compiler GHC findet das entsprechende Modul so jedoch nicht. Hier ist stattdessen folgender Befehl nötig:

```
import Control.Monad.Error
```

Um beide Haskell-Implementierungen problemlos nutzen zu können, behelfen wir uns einfach damit, eine neue Datei MonadError.hs in unserem Projektverzeichnis zu erstellen, in die wir die wichtigsten Definitionen aus den oben genannten Modulen hineinkopieren.

Betrachten wir nun die Definition der Klasse `MonadError`. Diese ist über einen Fehlertyp `e` und eine Monade `m` definiert:

```
class (Monad m) => MonadError e m | m -> e where
     throwError :: e -> m a
     catchError :: m a -> (e -> m a) -> m a
```

Für monadisches IO rufen wir dabei einfach die Funktionen `ioError` und `Catch` auf:

```
instance MonadError IOError IO where
     throwError = ioError
     catchError = catch
```

Für sonstige Berechnungen benötigen wir nun noch einen Datentyp, der das Ergebnis kapselt. Wir benutzen dafür einfach den Typ `Either`. Dieser ist in der Prelude wie folgt vordefiniert:

```
data Either a b = Left a | Right b
            deriving (Eq, Ord, Read, Show)
```

Die Idee ist nun, dass wir Fehler mit dem Konstruktor `Left` kapseln und erfolgreiche Berechnungen mit dem Konstruktor `Right`.

Um problemlos verschiedene Datentypen für Fehlermeldungen verwenden zu können, wird auch eine Klasse `error` definiert:

```
class Error a where
     noMsg  :: a
     strMsg :: String -> a

     noMsg    = strMsg ""
     strMsg _ = noMsg
```

Beispielsweise sind folgende Instanzen vordefiniert:

```
instance Error [Char] where
     noMsg  = ""
     strMsg = id

instance Error IOError where
     strMsg = userError
```

Nun können wir `Either e`, wobei `e` eine Instanz der Fehler-Klasse `Error` ist, zu einer Instanz von `MonadError` machen. Dazu müssen wir `Either e` überhaupt erst einmal zu einer Instanz von `Monad` machen:

```
instance (Error e) => Monad (Either e) where
     return        = Right
     Left  l >>= _ = Left l
     Right r >>= k = k r
     fail msg      = Left (strMsg msg)
```

Jetzt können wir wir `Either e` wie folgt zum Kapseln von möglicherweise fehlerhaften Berechnungen verwenden:

```
instance (Error e) => MonadError e (Either e) where
     throwError             = Left
     Left  l `catchError` h = h l
     Right r `catchError` _ = Right r
```

A.2 Fehlermeldungen in unterschiedlichen Sprachen

Um bei Fehlermeldungen leicht zwischen einer Ausgabe der Meldung in Deutsch oder einer Meldung in Englisch umschalten zu können, erweitern wir diese vordefinierten Klassen um weitere Klassen und Funktionen, so dass wir im Programm zunächst Fehlernummern verwenden können und dann erst bei der Ausgabe einer Fehlermeldung die entsprechende Fehlernummer in eine Fehlerbeschreibung in der gewünschten Sprache umgewandelt wird. Wir benötigen weitere Klassen, da wir teilweise zusätzliche Informationen berücksichtigen müssen, wie z.B. bei Parse-Fehlern die Position in der Eingabe-Datei.

Wir definieren zunächst den Typ für Fehlernummern wie folgt:

```
newtype ErrorNum = ErrNum Int
  deriving (Eq, Show)
```

Um die Fehlernummern sinnvoll zu nutzen, definieren wir entsprechende Konstanten:

```
errNumDefault :: ErrorNum
errNumDefault = ErrNum 0

errNumParseError :: ErrorNum
errNumParseError = ErrNum 1
```

(...)

Um nun den Fehlertyp `Either` verwenden zu können, müssen wir `ErrorNum` zu einer Instanz von `Error` machen, wir liefern dabei stets eine voreingestellte Fehlernummer zurück:

```
instance Error ErrorNum where
  noMsg = errNumDefault
  strMsg _ = noMsg
```

Für die Verwendung beliebiger Datentypen als Fehler-Information benötigen wir eine Klasse, die es ermöglicht, stets eine Fehlernummer zu erhalten:

```
class ErrorNumber e where
  errorNumber :: e -> ErrorNum
```

Natürlich soll auch der Fehlernummer-Typ eine Instanz dieser Klasse sein:

```
instance ErrorNumber ErrorNum where
  errorNumber = id
```

Nun benötigen wir eine Klasse, deren einzige Methode als Argument den Fehlertyp und eine Spracheinstellungs-Konstante erhält und daraus eine Fehlermeldung generiert:

```
class ErrorInfo e where
  errorMessage :: e -> MsgLanguage -> String
```

Die Spracheinstellungen definieren wir dazu wie folgt:

```
newtype MsgLanguage = MsgLang Int
     deriving (Eq, Show)

msgLDefault :: MsgLanguage
msgLDefault = msgLGerman

msgLGerman :: MsgLanguage
msgLGerman = MsgLang 1
```

```
msgLEnglish :: MsgLanguage
msgLEnglish = MsgLang 2
```

Somit können wir eine Funktion definieren, die für eine Fehlernummer eine entsprechende Meldung generiert:

```
errMsg num lang
  | num == errNumDefault && lang == msgLGerman = "Fehler!"
  | num == errNumDefault && lang == msgLEnglish = "Error!"
  | num == errNumParseError && lang == msgLGerman =
                                       "Fehler beim Parsen!"
  | num == errNumParseError && lang == msgLEnglish =
                                            "Parse Error!"
```

(...)

Wir können also nun für Instanzen der Klasse `ErrorNumber` folgende Hilfs-Funktion definieren, mit deren Hilfe sich auch Instanzen der Klasse `ErrorInfo` leicht definieren lassen:

```
errorMsg :: (ErrorNumber e) => e -> MsgLanguage -> String
errorMsg e = errMsg (errorNumber e)
```

Wir können z.B. wie folgt instanziieren:

```
instance ErrorInfo ErrorNum where
  errorMessage e l = errorMsg e l
```

Wir definieren nun folgende Typsynonyme:

```
type ErrResult = Either

type TiResult = ErrResult ErrorNum

type ParseResult = ErrResult PrsErrorInfo
```

Den Datentyp `PrsErrorInfo` definieren wir dabei wie folgt:

```
data PrsErrorInfo = PrsErrorInfo {
  prsErrorNum::ErrorNum,
  prsErrorPos::PrsPosition
} deriving (Eq,Show)
```

Eine Parse-Fehler-Information enthält also neben der Fehlernummer noch die Position des Schlüsselwortes oder Zeichens in der Eingabe, das den Fehler verursacht hat.

Den Datentyp `PrsErrorInfo` machen wir wie folgt zu einer Instanz der Klasse `ErrorInfo`:

```
instance ErrorInfo PrsErrorInfo where
  errorMessage e l = prsErrorMsgP (errorNumber e)
                                          (prsErrorPos e)  l

instance ErrorNumber PrsErrorInfo where
  errorNumber = prsErrorNum

prsErrorMsgP errnum pos language =
  prsErrorMsg errnum (prsPosFile pos) (prsPosLine pos)
(prsPosCol pos) language

prsErrorMsg errnum filename line col language =
  ("\"" ++ filename ++ "\"" ++ ":" ++ (show line) ++ "," ++
   (show col) ++ ": " ++
    (errMsg errnum language) ++ "\n")
```

Außerdem muss `PrsErrorInfo` eine Instanz der Klasse `Error` sein, um den Typ `ParseResult` als Instanz der Klasse `MonadError` nutzen zu können:

```
instance Error PrsErrorInfo where
  noMsg = PrsErrorInfo errNumDefault (PrsPosition "" 0 0)
  strMsg s = noMsg
```

Somit haben wir nun alle benötigten Datentypen und Klassen definiert. Zum Abschluss unserer Betrachtungen zur Ausgabe von Fehlermeldungen in unterschiedlichen Sprachen definieren wir noch ein paar einfache Hilfsfunktionen. Die Funktion `errResultOrError` kann dann dazu verwendet werden, um einen Fehler mit der Haskell-Funktion `error` zu erzwingen, falls die Berechnung fehlschlägt, und ansonsten einfach das Ergebnis ohne Kapselung zurückzuliefern:

```
errResultOk :: (ErrResult a b) -> Bool
errResultOk (Right _) = True
errResultOk _ = False

errGetInfo :: (ErrResult a b) -> a
errGetInfo (Left x) = x

errGetResult :: (ErrResult a b) -> b
errGetResult (Right x) = x

errGetMsg :: (ErrorInfo a) => (ErrResult a b) ->
                                        MsgLanguage -> String
errGetMsg x = errorMessage (errGetInfo x)

errResultOrError :: (ErrorInfo a) => (ErrResult a b) -> b
errResultOrError x = if errResultOk x then errGetResult x
else error (errGetMsg x msgLDefault)
```

Anhang B : Datenstrukturen von allgemeinem Interesse

Im Folgenden betrachten wir die für die Implementierung benötigten Datenstrukturen für Assoziationslisten, den Heap und den Stack. Die entsprechenden Definitionen sind angelehnt an [PJL91], aber an einigen Stellen erweitert und die abstrakten Datentypen werden mehr gekapselt, um so die Verwendung der Zugriffsfunktionen zu erzwingen und damit mehr Transparenz zu erreichen.

B.1 Assoziationslisten

Assoziationslisten dienen dazu, Elemente anhand eines Schlüssels zu finden. Der Einfachheit halber stellen wir dies durch eine Liste von Tupeln der Form (Schlüssel, Wert) dar:

```
newtype Assoc a b = Assoc [(a,b)]
 deriving (Show,Eq)
```

Hier wären jedoch bessere Suchstrukturen dazu geeignet, die Geschwindigkeit der Suche deutlich zu verbessern, so dass diese einfache Implementierung einen Ansatzpunkt für Optimierungen darstellt.

Betrachten wir nun die benötigten Operationen und ihre Implementierungen mit unserem einfachen Ansatz:

Zunächst müssen wir eine leere Assoziationsliste erstellen können:

```
assocEmpty :: Assoc a b
assocEmpty = Assoc []
```

Nachprüfen, ob die Liste leer ist, können wir mit folgender Funktion:

```
assocIsEmpty :: Assoc a b -> Bool
assocIsEmpty (Assoc xs) =
  case xs of
    [] -> True
    _  -> False
```

Die Anzahl der gespeicherten Elemente können wir wie folgt erhalten:

```
assocSize :: Assoc a b -> Int
assocSize (Assoc xs) = length xs
```

Wollen wir nun Elemente einfügen, so kann es sein, dass bereits ein Element mit gleichem Schlüssel in der Assoziationsliste enthalten ist. Wir stellen somit verschiedene Funktionen zur Verfügung:

1.) Falls bereits ein Element mit gleichem Schlüssel vorhanden ist, so führt dies zu einem Fehler:

```
assocInsertE :: (Eq a, Show a) => Assoc a b -> (a,b) ->
                                                  Assoc a b
assocInsertE cts@(Assoc xs) x =
     if (not (elem (fst x) (assocDomain cts))) then
          Assoc (x:xs)
     else
       error
        ("assocInsertE: Associationlist already contains " ++
         "an element with identification "
           ++ (show (fst x)) ++ "!")
```

2.) Es wird nur eingefügt, falls kein Element mit gleichem Schlüssel enthalten ist:

```
assocInsertNoUpdate :: Eq a => Assoc a b -> (a,b) ->
                                                    Assoc a b
assocInsertNoUpdate cts@(Assoc xs) x =
  if (not (elem (fst x) (assocDomain cts))) then
    Assoc (x:xs)
  else
    cts
```

3.) Falls ein Element mit gleichem Schlüssel enthalten ist, wird dieses anhand der noch zu definierenden Funktion `assocUpdate` überschrieben:

```
assocInsertUpdate :: Eq a => Assoc a b -> (a,b) -> Assoc a b
assocInsertUpdate cts@(Assoc xs) x =
     if (not (elem (fst x) (assocDomain cts))) then
          Assoc (x:xs)
     else
          assocUpdate cts (fst x) (snd x)
```

4.) Wenn wir sicher sein können, dass noch kein Element mit gleichem Schlüssel vorhanden ist, würden wir gerne direkt einfügen können, ohne die Elemente noch einmal zu überprüfen. Wir definieren daher:

```
assocInsertF :: Assoc a b -> (a,b) -> Assoc a b
assocInsertF (Assoc xs) x = Assoc (x:xs)
```

Natürlich darf diese Funktion nicht definiert werden, wenn bereits ein entsprechendes Element vorhanden ist, da dies sonst zu einer inkonsistenten Liste führt. Diese Funktion ist also mit Vorsicht zu genießen!

Definieren wir nun die benötigte Update-Operation:

Wir löschen dabei einfach das alte Element anhand der noch zu definierenden Funktion `assocRemove` und fügen das neue Element ein:

```
assocUpdate :: Eq a => Assoc a b -> a -> b -> Assoc a b
assocUpdate cts a n =
  let (Assoc xs) = (assocRemove cts a) in
    Assoc ((a,n):xs)
```

Um nun eine Liste mit Tupeln in die Funktion einfügen zu können (ggf. mit Update) definieren wir:

```
assocInsertUpdateList :: Eq a => Assoc a b -> [(a,b)] ->
                                                  Assoc a b
assocInsertUpdateList cts [] = cts
assocInsertUpdateList cts (x:xs) =
  (assocInsertUpdate (assocInsertUpdateList cts xs) x)
```

Die Lösch-Operation lässt sich leicht mit Hilfe von `filter` realisieren:

```
assocRemove :: Eq a => Assoc a b -> a -> Assoc a b
assocRemove (Assoc xs) a =
  Assoc (filter (\(x,_) -> (x /= a)) xs)
```

Betrachten wir nun die Nachschlage-Operationen:

Zunächst definieren wir eine Funktion `assocLookup`, die den zu einem Schlüssel passenden Wert nachschlägt. Wird kein Element mit diesem Schlüssel gefunden, so wird ein voreingestellter Wert, der als zusätzliches Argument übergeben werden muss, zurück geliefert:

```
assocLookup :: Eq a => Assoc a b -> a -> b -> b
assocLookup (Assoc xs) a def = assocLookup1 xs a def

assocLookup1 :: Eq a => [(a,b)] -> a -> b -> b
assocLookup1 [] _ def = def
assocLookup1 ((k,v):bs) k' def
     |k == k'  = v
     |otherwise = assocLookup1 bs k' def
```

Außerdem definieren wir eine Nachschlage-Operation, die die Rückgabe mit dem Datentyp `Maybe` kapselt, d.h. es wird `Nothing` zurückgeliefert, falls kein entsprechendes Element gefunden werden kann. Es erweist sich als nützlich, nicht nur eine Suche über den Schlüssel, sondern auch über den Wert zuzulassen:

```
assocLookElemFst :: Eq a => Assoc a b -> a -> Maybe (a,b)
assocLookElemFst (Assoc xs) a =
  let xs' = dropWhile (\(x,_)-> x /= a) xs in
    case xs' of
      [] -> Nothing
      (x:xs) -> Just x
```

```
assocLookElemSnd :: Eq b => Assoc a b -> b -> Maybe (a,b)
assocLookElemSnd (Assoc xs) e =
  let xs' = dropWhile (\(_,x)-> x /= e) xs in
    case xs' of
      [] -> Nothing
      (x:xs) -> Just x
```

Abschließend definieren wir noch zwei Funktionen, um alle Schlüssel oder alle Werte erhalten zu können:

```
assocDomain :: Assoc a b -> [a]
assocDomain (Assoc alist) = [key|(key,_)<-alist]

assocRange :: Assoc a b -> [b]
assocRange (Assoc alist) = [val|(_,val)<-alist]
```

B.2 Heap

Der Heap speichert beliebige Elemente, wobei auf die Speicherplätze anhand von Heap-Adressen zugegriffen wird. Heap-Adressen definieren wir dabei wie folgt:

```
newtype HeapAddress = HeapAddress {
  heapAddress :: Int
} deriving (Eq,Ord)
```

Den Heap-Speicher können wir somit als Assoziationliste mit Heap-Adressen als Schlüssel darstellen, d.h. mit dem Typ `(Assoc HeapAddress a)`. Zusätzlich zum eigentlichen Speicherplatz enthält unser Heap-Datentyp auch noch eine Zahl vom Typ `Int`, welche die Anzahl der gespeicherten Objekte angibt, und eine Liste mit freien Adressen. Wir definieren also:

```
data Heap a = Heap Int [HeapAddress] (Assoc HeapAddress a)
     deriving (Eq)
```

Nun können wir die Operationen auf diesem Datentyp definieren:

Zunächst benötigen wir eine Funktion, die einen neuen Heap initialisiert:

```
heapInit :: Heap a
heapInit = Heap 0 [(HeapAddress x) | x <- [1..]] assocEmpty
```

Danach benötigen wir natürlich eine Einfüge-Operation:

```
heapInsert :: Heap a -> a -> (Heap a, HeapAddress)
heapInsert (Heap size (next:free) cts) n =
  ((Heap (size+1) free (assocInsertF cts (next,n))),next)
```

Da wir auch gerne ganze Listen von zu speichernden Objekten auf einmal einfügen möchten, definieren wir auch eine Funktion `heapInsertList`:

```
heapInsertList :: Heap a -> [a] -> (Heap a,[HeapAddress])
heapInsertList heap xs = let (h,as) = heapInsertList1
(heap,[]) xs in (h,(reverse as))

heapInsertList1 :: (Heap a,[HeapAddress]) -> [a] ->
                                        (Heap a,[HeapAddress])
heapInsertList1 (heap,addrs) [] = (heap,addrs)
heapInsertList1 ((Heap size (next:free) cts),addrs) (x:xs) =
  heapInsertList1 ((Heap (size+1) free
                 (assocInsertF cts (next,x))),next:addrs) xs
```

Außerdem definieren wir eine Update-Funktion:

```
heapUpdate :: Heap a -> HeapAddress -> a -> Heap a
heapUpdate (Heap size free cts) a n =
  let (Assoc cts') = (heapRemove cts a) in
   (Heap size free (Assoc ((a,n):cts')))
```

Selbstverständlich soll man Elemente auch wieder löschen können:

```
heapDelete :: Heap a -> HeapAddress -> Heap a
heapDelete (Heap size free cts) a =
  (Heap (size-1) (a:free) (remove cts a))

heapRemove :: (Assoc HeapAddress a) -> HeapAddress ->
  (Assoc HeapAddress a)
heapRemove (Assoc []) a =
  error ("heapRemove: Trying to update/delete element at" ++
         "non-existing address "++"#"++(show heapAddress a)++
         "!")
heapRemove (Assoc ((a',n):cts)) a
  |a == a' = (Assoc cts)
  |otherwise =
    let (Assoc cts') = (heapRemove (Assoc cts) a) in
      Assoc ((a',n):cts')
```

Eine Nachschlage-Funktion darf natürlich auch nicht fehlen:

```
heapLookup  :: Heap a -> HeapAddress -> a
heapLookup (Heap size free cts) a =
 assocLookup cts a (error ("heapLookup: Address "++
                   "#"++(show heapAddress a)++" is unused!"))
```

Zu guter Letzt definieren wir noch zwei Funktionen, die die Liste der freien Adressen bzw. die Anzahl der gespeicherten Objekte zurückliefern:

```
heapAddresses :: Heap a -> [HeapAddress]
heapAddresses (Heap _ _ (Assoc cts)) =
                                    [ addr| (addr,node)<-cts]
```

```
heapSize :: Heap a -> Int
heapSize (Heap size _ _)= size
```

B.3 Stack

In [PJL91] wird der Stack einfach in einer Listendarstellung verwendet und direkt auf die Elemente der Liste zugegriffen. Wir kapseln diese Liste hier jedoch in einem abstrakten Datentyp:

```
newtype Stack a = Stack [a]
     deriving (Eq)
```

Anstelle des direkten Zugriffs auf die Liste erzwingen wir somit die Verwendung unserer Zugriffs-Funktionen. Neben den bekannten Stack-Funktion `Push` und `Pop`, mit denen ein Element auf den Stack gelegt bzw. das obere Element aus dem Stack zurückgeliefert (und entfernt) wird, unterstützen wir die Operationen `Look` und `Drop`, wobei mit `Look` das obere Element ohne dessen Entfernung nachgeschlagen werden kann, und `Drop` das obere Element entfernt, ohne dieses zurückzuliefern. Außerdem implementieren wir je eine Funktion zur Initialisierung eines Stacks, zur Abfrage, ob der Stack leer ist und zur Überprüfung, ob genau ein Element auf dem Stack liegt. Betrachten wir nun die Funktionen im Einzelnen:

Einen neuen Stack initialisieren:

```
stackInit :: Stack a
stackInit = (Stack [])
```

Überprüfen, ob der Stack leer ist:

```
stackIsEmpty :: Stack a -> Bool
stackIsEmpty (Stack []) = True
stackIsEmpty _ = False
```

Überprüfen, ob genau ein Element auf dem Stack liegt:

```
stackOneLeft :: Stack a -> Bool
stackOneLeft (Stack [_]) = True
stackOneLeft _ = False
```

Die `Look`-Operation:

```
stackLook (Stack []) = error ("Stack ist leer!")
stackLook (Stack (x:xs)) = x
```

Die `Pop`-Operation:

```
stackPop :: Stack a -> (Stack a, a)
stackPop (Stack []) = error ("Stack ist leer!")
stackPop (Stack (x:xs)) = ((Stack xs),x)
```

Die `Push`-Operation:

```
stackPush :: Stack a -> a -> Stack a
stackPush (Stack xs) x = (Stack (x:xs))
```

Die `Drop`-Operation:

```
stackDrop :: Stack a -> Int -> Stack a
stackDrop (Stack xs) n = (Stack (drop n xs))
```

Literaturverzeichnis

[Abr90] ABRAMSKY, SAMSON: *The lazy lambda calculus*, In: TURNER, DAVID A. (Herausgeber): *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, Kapitel 4, Seiten 65 - 116, Addison-Wesley, 1990

[AF97] ARIOLA, ZENA M.; FELLEISEN, MATHIAS: *The call-by-need lambda calculus*, Journal of Functional Programming, Band 7, Nr. 3, Seiten 265 - 301, Cambridge University Press, 1997

[AFM+95] ARIOLA, ZENA M.; FELLEISEN, MATHIAS; MARAIST, JOHN; ODERSKY, MARTIN; WADLER, PHILLIP: *A Call-By-Need Lambda Calculus*, Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 23-25, 1995, Seiten 233 - 246, ACM Press, 1995

[Bar84] BARENDREGT, HENDRIK PIETER: *The Lambda Calculus, Its Syntax and Semantics*, Elsevier Science Publishers, 1984

[Bir98] BIRD, RICHARD: *Introduction to Functional Programming using Haskell*, Prentice-Hall Europe, 1998

[Dyb96] DYBVIG, R. KENT: *The Scheme Programming Language, Second Edition*, Prentice Hall, 1996

[Gor94] GORDON, ANDREW D.: *A Tutorial on Co-induction and Functional Programming*, Proceedings of the 1994 Glasgow Workshop on Functional Programming, Seiten 78-95, Springer Verlag, 1994

[How96] HOWE, DOUGLAS J.: *Proving Congruence of Bisimulation in Functional Programming Languages*, Information and Computation, Band 124, Nr. 2, Seiten 103 - 112, Elsevier Science Publishers, 1996

[Hug89/90] HUGHES, JOHN: *Why Functional Programming Matters*, The Computer Journal, Band 32, Nr. 2, 1989, Seiten 98 - 107, Oxford University Press, 1989 bzw. David A. Turner (Herausgeber): Research Topics in Functional Programming, Seiten 17 - 42, Addison-Wesley, 1990

[KOR93] KLOP, JAN W.; VAN OOSTROM, VINCENT; VAN RAAMSDONK, FEMKE: *Combinatory Reduction Systems: introduction and survey*, Theoretical Computer Science, Band 121, Seiten 279 - 308, Elsevier Science Publishers, 1993

[Man99] MANN, MATTHIAS: *Gleichheitsanalyse von Ausdrücken in nicht-strikten funktionalen Programmiersprachen unter Verwendung der Kontextanalyse*, Diplomarbeit, Johann Wolfgang Goethe-Universität Frankfurt am Main, 1999

[MG01] MARLOW, SIMON; GILL, ANDY: *Happy User Guide*, 2001, nur im Internet verfügbar: http://www.haskell.org/happy/

[MN98] MAYR, RICHARD; NIPKOW, TOBIAS: *Higher-Order Rewrite Systems and their Confluence*, Theoretical Computer Science, Band 192, Nr. 1, Seiten 3 - 29, Elsevier Science Publishers, 1998

[MOW98] MARAIST, JOHN; ODERSKY, MARTIN: *The call-by-need lambda calculus*, Journal of Functional Programming, Band 8, Nr. 3, Seiten 275 - 317, Cambridge University Press, 1998

[MTH+97] MILNER, ROBIN; TOFTE, MADS; HARPER, ROBERT; MACQUEEN, DAVID: *The Definition of Standard ML - Revised*, MIT, 1997

[Nip93] NIPKOW, TOBIAS: *Orthogonal Higher-Order Rewrite Systems are Confluent*, M. Bezem und J.F. Groote (Herausgeber): Proceedings of the International Conference on Typed Lambda Calculi and Applications, Lecture Notes in Computer Science 664, Seiten 306 - 317, Springer Verlag, 1993

[PJ98] PEYTON JONES, SIMON: *A short introduction to Haskell and its advantages*, 1998, nur im Internet verfügbar: http://www.haskell.org/aboutHaskell.html

[PJ03] PEYTON JONES, SIMON: *Haskell 98 Language and Libraries – The Revised Report*, Cambridge University Press, 2003

[PJL91] PEYTON JONES, SIMON; LESTER, DAVID R.: *Implementing Functional Languages: a Tutorial*, Prentice-Hall International, 1991

[Plo77] PLOTKIN, GORDON D.: *LCF as a programming language*, Theoretical Computer Science, Band 5, Seiten 223 - 257, Elsevier Science Publishers, 1977

[PvE93] PLASMEIJER, RINUS; VAN EEKELEN, MARKO: *Functional programming and parallel graph rewriting*, International Computer Science Series, Addison Wesley, 1993

[San97] SANDS, DAVID: *From SOS Rules to Proof Principles : An Operational Metatheory for Functional Languages*, Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997, Seiten 428 - 441, ACM Press, 1997

[Sch97] SCHMIDT-SCHAUSS, MANFRED: *Praktische Informatik 1, WS 97/98*, Manuskript der Vorlesung, Johann Wolfgang Goethe-Universität Frankfurt am Main, 1997/98,
http://www.ki.informatik.uni-frankfurt.de/lehre/infoI/main.html

[Sch00] SCHMIDT-SCHAUSS, MANFRED: *Funktionale Programmierung I, Sommersemester 2000*, Manuskript der Vorlesung, Johann Wolfgang Goethe-Universität Frankfurt am Main, 2000,
http://www.ki.informatik.uni-frankfurt.de/lehre/fpI/main.html

[Sch01] SCHMIDT-SCHAUSS, MANFRED: *Funktionale Programmierung II, Sommersemester 2001*, Manuskript der Vorlesung, Johann Wolfgang Goethe-Universität Frankfurt am Main, 2001,
http://www.ki.informatik.uni-frankfurt.de/lehre/SS2001/FP-II/main.html

[Sco69/93] SCOTT, DANA S.: *A type-theoretical alternative to CUCH, ISWIM and OWHY*, Theoretical Computer Science, Band 121, Seiten 411 - 440, Elsevier Science Publishers, 1993. Neudruck eines 1969 erstellten Manuskripts.

[SSS01/03] SCHMIDT-SCHAUSS, MANFRED; STUBER, JÜRGEN: *On the complexity of linear and stratified context matching problems*, INRIA, Rapport de recherche RR-4923, 2003 bzw. Frank-14, 2001,
http://www.ki.informatik.uni-frankfurt.de/papers/articles.html

[Tak95] TAKAHASHI, MASAKO: *Parallel reduction in λ-calculus*, Information and Computation, Band 118, Seiten 120 - 127, Elsevier Science Publishers, 1995

[Tho97] THOMPSON, SIMON: *Higher Order + Polymorphic = Reuseable*, 1997, nur im Internet verfügbar: http://www.cs.kent.ac.uk/pubs/1997/224/index.html

[Tur95] TURNER, DAVID A.: *Miranda: a non-strict functional language with polymorphic types*, Jean-Pierre Jouannaud (Herausgeber): Functional Programming Languages and Computer Architecture, Nancy, France, September 16-19, 1985, Proceedings. Lecture Notes in Computer Science 201, Seiten 1 - 16, Springer Verlag, 1985

[Wad95] WADLER, PHILIP: *Monads for Functional Programming*, In „Advanced Functional Programming", Springer Verlag, 1995