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

Diplomarbeit

Erweiterung der Kontextanalyse für nicht-strikte funktionale Programmiersprachen um Funktionskontexte

Dirk Rehberger

11. November 1999

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

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

Frankfurt am Main, 11. November 1999

Dirk Rehberger

Inhaltsverzeichnis

1 Einführung **5**

2 Funktionales Programmieren **7**
2.1 Grundsätzliches . . . 7
2.2 Referentielle Transparenz . . . 8
2.3 Auswertung . . . 8
2.4 Haskell . . . 9

3 Die Kernsprache Λ **12**
3.1 Syntax . . . 12
3.1.1 Typsystem und alternatives `case` . . . 13
3.2 Operationale Semantik . . . 14
3.2.1 Reduktion . . . 14
3.2.2 Abstraktion von Λ . . . 15
3.2.3 Die schwache Kopfnormalform . . . 15
3.3 Wohlgetyptheit . . . 16
3.4 Kontextuelle Ordnung . . . 17
3.5 club und lub . . . 20

4 Kontexte **23**
4.1 Syntax . . . 23
4.2 Joint-evaluation-Kontexte . . . 25
4.3 Semantik . . . 25
4.3.1 Primitive Konkretisierung von Kontexten . . . 25
4.3.2 Konkretisierung von Kontexten . . . 30
4.4 Funktionskontexte . . . 31
4.5 Konstruktion von η . . . 35
4.6 Weitere Sätze über η und γ . . . 38
4.7 Projektionen . . . 39
4.7.1 Überblick . . . 40
4.7.2 Grundlegende Definitionen . . . 40
4.7.3 Syntaktische Projektionen auf S-strukturierten Kontexten . . . 41
4.7.4 Großzügige und kleinliche Projektionen . . . 42
4.7.5 Projektionen auf `where`-Kontexten . . . 45
4.8 Komplemente . . . 48

5 Kalkül **49**
5.1 Kontextanalyse . . . 49
5.1.1 Tableaus . . . 50
5.1.2 Regeln . . . 52
5.2 Neue Regeln für die Behandlung von Funktionen . . . 54
5.2.1 Regel zur Anwendungsauflösung . . . 54

5.2.2 Regel für Konstruktor mit Kontextvariablen 56
5.2.3 Regel für `bot` mit Kontextvariablen 57

6 Analysen mit Funktionskontexten **59**
6.1 Analyse von $f \in C_1 \overset{\forall}{\to} C_2$ 59
6.1.1 Beispiele und ein Verfahren für einfache Fälle 60
6.1.2 Verallgemeinerung des Verfahrens 62
6.1.3 Korrektheit 67
6.1.4 Vollständigkeit 72
6.2 Analyse von $f \in C_1 \overset{\exists}{\to} C_2$ 74
6.2.1 Korrektheit und Vollständigkeit des Verfahrens 75

7 Implementierung **76**
7.1 Kontexte 76
7.1.1 Darstellung 76
7.1.2 Monadisches Programmieren 78
7.1.3 Die Monade für Kontextdefinitionen 79
7.1.4 Werkzeuge 83
7.1.5 Inklusion, Konsistenz und Komplemente 88
7.1.6 Projektionen 89
7.2 Kernsprache 92
7.3 Tableaus 93
7.3.1 Arbeitsgänge 93
7.3.2 Ein einzelnes Tableau 95
7.4 Überlegungen zur Tableau-Organisation 95
7.4.1 Einbinden der Analysen höherer Ordnung in den Tableaukalkül 95
7.4.2 Die konjunktive Struktur der Knoten 97
7.5 Ausfüllen des `TableauSet`-Gerüsts 98
7.5.1 Die verwendeten Typen für die Tableaus 98
7.5.2 Die Tableauregeln für die Behandlung höherer Ordnung 99
7.5.3 Heuristik, Lösung und Aufruf der Phasen 102
7.6 Analysen höherer Ordnung 104
7.6.1 Die $\overset{\exists}{\to}$-Analyse 104
7.6.2 Die $\overset{\forall}{\to}$-Analyse 105
7.7 Lösungen 111
7.8 Andere Module 115
7.8.1 Das Hauptprogramm 115
7.8.2 Mengen 116

8 Ergebnisse **117**
8.1 Reine Tableauanalysen 117
8.2 Ergebnisse von Analysen höherer Ordnung 119

9 Zusammenfassung und Ausblick **122**
9.1 Zusammenfassung 122
9.2 Ausblick 122

A Projektionen auf `where`-Kontexten: weitere Fälle **124**

B Einige Definitionen von Kernsprachenausdrücken **128**

Kapitel 1

Einführung

Tests sind ein integraler Bestandteil jeder Softwareentwicklung. Denn Fehler beim Programmieren sind unvermeidlich. Um zu sehen, ob ein Programm tatsächlich die gewünschte Funktionalität bietet, muß es daher im Laufe seiner Entstehung ständig und in allen seinen Teilen getestet werden.

Ein Test im engeren Sinne besteht darin, ein Programm auf verschiedenen Eingaben — insbesondere solchen, die man als kritisch erachtet — ablaufen zu lassen und die jeweilige Ausgabe mit dem gewünschten Resultat zu vergleichen.

Regelmäßig stellt sich dabei die Frage, für welche Eingaben eine bestimmte Ausgabe produziert wird. Z.B. hat man einen Fehler entdeckt und interessiert sich dafür, unter welchen Umständen er zustandekommt. In solchen Fällen ist es nützlich, ein Werkzeug zu haben, das aus einer gegebenen Ausgabe die dazugehörigen, kritischen Eingaben automatisch konstruiert.

Für den Sprachkern sogenannter *nicht-strikter funktionaler Programmiersprachen* leistet dies die *Kontextanalyse*. Sie wurde in [SSS97a] vorgestellt und wird derzeit in [Sch99b] weiterentwickelt.

Die dort benutzten Werte für Ein- und Ausgaben sind keine konkreten Ausdrücke der Programmiersprache, sondern Abstraktionen, die sogenannten *Anforderungskontexte* oder kurz *Kontexte*[1]. Diese sind eine Repräsentation von Mengen von Programmausdrücken. Man möchte ja die Möglichkeit haben, ganze Klassen von Ausdrücken unter einem Begriff zusammenzufassen: So bezeichnet z.B. der Kontext `Top` alle möglichen Ausdrücke, `Bot` steht (im wesentlichen) für alle solchen Ausdrücke, deren Auswertung nicht terminiert (d.h. deren Wert unbestimmt ist), und `Inf` steht für alle Listen, die unendlich lang sind oder deren Ende einen unbestimmten Wert hat. Große wie kleine Mengen von Ausdrücken lassen sich so darstellen.

Denn wenn auch ein Kontext immer für eine Menge konkreter Ausdrücke steht, so ist diese Abstraktion doch relativ fein strukturiert. Es gibt unendlich viele Kontexte, die sich in ihren repräsentierten Ausdrücken nicht überschneiden. Für jedes Datenobjekt a, das aus beliebigen Datenkonstruktoren oder Ausdrücken mit unbestimmtem Wert aufgebaut ist, gibt es sogar immer einen Kontext, der genau die zu a verhaltensgleichen Ausdrücke darstellt.

Demgegenüber steht die Tatsache, daß alle Ausdrücke, die als Funktion angesehen werden, nur durch einen einzigen Kontext, `Fun`, repräsentiert werden.

Nun sind in funktionalen Sprachen *Funktionen höherer Ordnung* allgegenwärtig; dabei handelt es sich um Funktionsausdrücke, die selbst eine Funktion als Argument erwarten (ein „Argument höherer Ordnung“) oder eine Funktion als Ergebnis liefern. Das bedeutet, daß Funktionen ganz gewöhnliche Ausdrücke sind wie andere auch und an vielen Stellen eines Programms auftauchen. Daher scheint bezüglich `Fun` eine feinere Gliederung der Kontexte angemessen zu sein.

Zwar kann die Kontextanalyse auch Funktionen höherer Ordnung bearbeiten. Z.B. löst sie die Anfrage `map` $f\ xs \in$ `Inf`, wobei f ein Argument höherer Ordnung ist. Dies ist aber nur möglich, weil bei dieser Anfrage f aufgrund der nicht-strikten Semantik der Kernsprache nicht ausgewertet wird und somit nicht näher als durch `Top` beschrieben werden muß. Eine weitergehende Analyse auf einer Funktion höherer Ordnung ist mit den bisherigen Kontexten nicht möglich.

[1] In [Sch99b] wird inzwischen die Bezeichnung *demands*, zu deutsch *Anforderungen* verwendet. Gemeint ist aber das gleiche wie hier, nämlich Anforderungskontexte.

Die vorliegende Arbeit befaßt sich mit der oben angesprochenen Aufgabe, an der Stelle von Fun feiner zu abstrahieren, und führt dazu Kontexte der Form $A \xrightarrow{\forall} B$ ein. Die Kontextanalyse wird um eine Behandlung dieser Terme erweitert. Das ganze geschieht zunächst theoretisch und wird im Anschluß daran um eine Implementierung in der funktionalen Sprache Haskell ergänzt.

Die Arbeit ist im einzelnen wie folgt gegliedert:

Kapitel 2 gibt eine Einführung in die Prinzipien funktionaler Programmiersprachen. Sowohl die Sprache der zu analysierenden Programme als auch die, in der die Analyse implementiert ist, gehören ja zu dieser Klasse von Sprachen, wenn auch auf unterschiedlichem Niveau. Die letztere der beiden, Haskell, wird in einem eigenen Abschnitt vorgestellt.

Die Kernsprache behandelt Kapitel 3. Neben den nötigen Definitionen geht es dort um die Herleitung einiger Eigenschaften, die für die Kontexte und die Analyse verwendet werden.

In Kapitel 4 werden Kontexte definiert. Die Funktionskontexte werden in Abschnitt 4.4 eingeführt. Weiterhin werden einige für die Analyse benötigte Sätze bewiesen.

Kapitel 5 behandelt die Kontextanalyse aus [Sch99b]. Zusätzlich werden bereits Regeln definiert, die mit Argumenten höherer Ordnung umgehen können, und Sätze über deren Korrektheit und Vollständigkeit bewiesen.

Wird einer der neu eingeführten Funktionskontexte schon in einer Anfrage an die Kontextanalyse erwähnt, so kann dies — außer in einfachen Fällen — erst mit einem der Analyseverfahren verarbeitet werden, die in Kapitel 6 vorgestellt werden. Dort wird zum einen die Methodik erörtert. Zum anderen werden die Verfahren auf Korrektheit und Vollständigkeit untersucht.

Die Implementierung der so erweiterten Kontextanalyse wird in Kapitel 7 vorgestellt. Sie erfolgt in Haskell 98, der zur Zeit der Anfertigung dieser Arbeit aktuellen Version der Programmiersprache Haskell.

In Kapitel 8 werden einige Beispielanalysen dargestellt und besprochen.

Kapitel 9 enthält schließlich eine Zusammenfassung dieser Arbeit sowie einen Ausblick auf künftige Ergänzungsmöglichkeiten.

Kapitel 2

Funktionales Programmieren

„Rekursionen! Igitt, die sind doch so ineffizient.“ Derartige Aussprüche bekommt man bisweilen über funktionale Programmiersprachen zu hören. In der Tat sind Rekursionen eines der wichtigsten Ausdrucksmittel in diesen Sprachen. Wer so etwas sagt, vergißt aber, daß sich der Ablauf eines funktionalen Programms grundsätzlich unterscheidet von der Ausführung etwa von C- oder Java-Code. Z.B. entfällt das Anlegen eines Aufruf-Stacks für einen Funktionsaufruf, an das die Urheber solcher Aussagen wohl denken und wie es in imperativen Sprachen erforderlich ist, in funktionalen Sprachen komplett.

Wenn funktionale Programme trotzdem nicht die Ablaufgeschwindigkeit ihrer imperativen Gegenstücke erreichen, so liegt das sicher auch daran, daß die ersteren auf einer logischen Ebene geschrieben sind, die weiter von der Hardware entfernt ist als bei letzteren.

Für viele Zwecke wichtiger ist jedoch die andere Seite der Medaille: daß man nämlich mit funktionalen Sprachen deutlich strukturierter und klarer programmieren kann. Die Möglichkeit, daß Argumente oder Ergebnisse von Funktionen wiederum Funktionen sein können, erlaubt eine weitestgehende Modularisierung auf allen Ebenen. Programme lassen sich nicht nur leichter lesen, sondern auch leichter schreiben und wiederverwenden. Das ist wiederum ein Vorteil zwar nicht für ihre Laufzeit, aber für die Zeit, die mit ihrer Entwicklung zugebracht wird.

In diesem Kapitel werden die Wesenszüge der funktionalen Programmierung umrissen. Ein eigener Abschnitt beschäftigt sich mit Haskell, der Sprache, in der der Code dieser Arbeit geschrieben ist.

2.1 Grundsätzliches

Funktionale Sprachen gehören zur Familie der *deklarativen Programmiersprachen.* Diese zeichnen sich vage formuliert dadurch aus, daß man, um ein Problem zu lösen, die Beschreibung des Problems angibt anstelle des genauen Lösungsweges. Dazu ist ein anderes Denken erforderlich als in den verbreiteten, imperativen Sprachen; die Programme sind *abstrakter.*

Ein imperatives Programm setzt sich im Prinzip aus einer Folge von Anweisungen zusammen. Der Ablauf eines Programms besteht aus einer sequentiellen Abarbeitung dieser Kommandos. Die Anweisungen können im allgemeinen auf einen globalen Zustand lesend und schreibend zugreifen, der Variablenwerte gespeichert hält. Eine Variable steht also für einen Speicherplatz im globalen Zustand, der Daten enthält; diese Daten können während des Programmablaufs geändert werden.

Dieses Vorgehen ist sehr hardwarenah, d.h. es entspricht den Ablaufstrukturen, die auf der heute gängigen Hardware möglich sind, und kann damit effizient in computerinterne Instruktionen übersetzt werden.

Ein funktionales Programm hingegen besteht im Prinzip aus einer Menge von Funktionsdefinitionen. Ein Programm läuft ab, indem ein ausgewählter Ausdruck solange *reduziert* wird, bis sich dessen Wert ergibt. Die Reduktion besteht prinzipiell jeweils aus der Anwendung einer Funktion auf Argumente. Einen globalen Zustand gibt es nicht; die Unterscheidung zwischen Daten

und Prozeduren entfällt. Es gibt zwar für gewöhnlich auch Datenobjekte, welche aber ebenfalls als die Anwendung einer Funktion (eines „Konstruktors“) auf Argumente zu betrachten sind, nur daß diese keine Definition haben und daher nicht weiter reduzieren. Variablen stehen für formale Argumente von Funktionen, die während des Programmablaufs einen einmal zugewiesenen Wert nicht mehr ändern.

Z.B. läßt sich der Quicksort-Algorithmus in Haskell folgendermaßen schreiben ([BW88]):

```
qsort []     = []
qsort (x:xs) = qsort [u | u <- xs, u < x] ++ [x] ++
               qsort [u | u <- xs, u > x]
```

An dem Programm ist weniger der konkrete Lösungsweg, aber deutlich die Spezifikation des Quicksort-Prinzips zu erkennen. Es ist relativ abstrakt formuliert; das ist weiter von der Hardwareebene entfernt, aber dafür liegt es der menschlichen Denkweise näher.

Funktionale Sprachen gehen auf den Λ-Kalkül zurück (s. z.B. [Bar84]). Dieser ist eine mathematische Formalisierung von Berechnungen, die mittels Funktionsanwendungen geführt werden. Dieses Vorhandensein einer mathematischen Grundlage drückt sich in der *referentiellen Transparenz* aus:

2.2 Referentielle Transparenz

Weil ein globaler Zustand fehlt, können natürlich keinerlei Seiteneffekte auf ihn oder durch ihn ausgeübt werden. Andersherum betrachtet heißt dies, daß der Wert einer Funktionsanwendung ausschließlich von den Werten der Funktionsargumente abhängt. Diese Eigenschaft wird als referentielle Transparenz bezeichnet und macht funktionale Sprachen relativ leicht einer mathematischen Behandlung zugänglich wie z.B. einer Korrektheitsüberprüfung oder auch der Kontextanalyse, was in imperativen Sprachen ungleich schwieriger ist. Die oben erwähnte Unabänderlichkeit des Wertes, für den eine Variable steht, ist eine Folge der referentiellen Transparenz.

2.3 Auswertung

Wie schon gesagt, wird ein Ausdruck mittels fortgesetzter Reduktion ausgewertet. Stoppt dieser Vorgang, ist mit dem Resultat der Wert des Ausdrucks gegeben. Der Wert eines Ausdrucks ohne terminierende Reduktion ist nicht definiert.

Eine Reduktion überführt einen Teilausdruck, der eine Funktionsanwendung darstellt, innerhalb des Gesamtausdrucks in das entsprechende, durch die Funktionsdefinition ermittelte Reduktionsergebnis. Ein Teilausdruck, der so behandelt werden kann, heißt *Redex*.

Nun kann ein Ausdruck mehrere Redexe enthalten. Möglicherweise ist er selbst einer oder auch einige seiner Unterausdrücke. Jedenfalls kann es vorkommen, daß man sich zwischen verschiedenen Redexen entscheiden muß. Für solche Fälle muß man *Auswertungsstrategien* benutzen:

- Die **Normalordnung** entscheidet sich wann immer möglich dafür, einen ganzen Ausdruck zu reduzieren anstatt einen seiner Teilausdrücke. („Äußerster Redex zuerst“)
- Dagegen reduziert die **Anwendungsordnung** erst alle Argumente einer Funktion, bevor es die Gesamtanwendung selbst reduziert. („Innerster Redex zuerst“)

Aus diesen beiden Extremen lassen sich auch gemischte Strategien zusammensetzen. Eine zentrale Eigenschaft der funktionalen Sprachen ist es nun, daß, Terminierung vorausgesetzt, das Ergebnis unabhängig von der Auswertungsreihenfolge ist ([Bar84]).[1] Damit weisen sie eine inhärente Parallelisierbarkeit auf, was sehr zukunftsträchtig erscheint.

[1] Diese Eigenschaft gilt nur für *pure* funktionale Sprachen, also solche, die — wie hier beschrieben — keine Seiteneffekte zulassen. Daneben gibt es auch Sprachen, die dies erlauben, wie z.B. Scheme.

2.4 Haskell

Haskell ist ein Vertreter der *lazy* (englisch: faul) oder *verzögert auswertenden* funktionalen Sprachen. Das bedeutet nichts anderes, als daß dort die Normalordnung realisiert wird. Die Bezeichnung *nicht-strikt* dafür ist ebenfalls üblich, da Striktheit der Auswertung in Anwendungsordnung entspricht.

Ein Vorteil verzögerter Auswertung ist, daß auch mit unendlichen Objekten umgegangen werden kann, oder genauer gesagt mit Objekten, deren vollständige Auswertung unendlich viele Reduktionen beanspruchen würde. Wenn nämlich an einer beliebigen Stelle im Programmablauf klar ist, daß das Objekt nicht mehr zur Weiterverarbeitung benötigt wird, dann wird es einfach „weggeworfen“ und nicht mehr beachtet. In einer strikten Sprache würden hingegen Anwendungen auf unendlichen Objekten nie terminieren können. Diese Eigenschaft wird z.B. bei den in dieser Arbeit verwendeten Tableaus benutzt, die potentiell unendlich groß sind.

Haskell unterscheidet sich von der wohl bekanntesten funktionalen Sprache, Lisp, nicht nur in der verzögerten Auswertung, sondern auch stark in der Syntax: Es gibt z.B. Infixnotation, algebraische Datentypen, mehr syntaktischen Zucker, und man benötigt keine exzessiven Klammerungen.

Wie [PJHA+99] zu entnehmen ist, liegen die Anfänge von Haskell in 1987. Seitdem wurde die Sprache in mehreren Schritten weiterentwickelt und ist zu einem Standard unter den nicht-strikten funktionalen Sprachen geworden. Die aktuelle Version ist Haskell 98; sie wird in [PJHA+99] vollständig festgelegt. Wir stellen daraus anhand eines kurzen Beispielprogramms einige ihrer Charakteristika vor, die der Programmcode dieser Arbeit benutzt.

```
> module HaskellBeispiel
> where
```

Diese Zeilen stellen die Deklaration eines *Moduls* dar, der obersten Hierarchiestufe eines Haskell-Programms. Der Code in den meisten der einzelnen numerierten Abschnitte in Kapitel 7 ist jeweils in einem eigenen Modul untergebracht. Das Importieren anderer Module sowie das Exportieren ausgewählter Funktionen und Typen sind möglich, in diesem Beispielprogramm aber nicht vonnöten.

Jede Programmzeile beginnt mit einem `>`. Damit wird die Möglichkeit genutzt, im *literate style* zu programmieren. Dies bedeutet, daß alle Zeilen, die nicht mit diesem Zeichen beginnen, als Kommentare betrachtet werden. So z.B. auch der Text dieses Abschnittes; er befindet sich zusammen mit dem Haskell-Code in einer Datei, die daher eine gültige Eingabe für den Compiler darstellt.

```
> data Tree a = Leaf a
>             | Branch a (Tree a) (Tree a)
```

Diese kompakte Definition erklärt einen *algebraischen Datentyp* `Tree` mit den *Konstruktoren* `Leaf` und `Branch`. Ausdrücke dieses Typs repräsentieren binäre Bäume, die in ihren Knoten eine Marke vom Typ `a` tragen. `a` kann als Typvariable jeden beliebigen Typ annehmen — man sagt, `Tree` ist *polymorph* über dem Typ der Knotenmarkierung definiert. Z.B. steht der Typ `Tree String` für Bäume, deren Knoten mit Zeichenketten markiert sind.

Eine alternative Methode, einen algebraischen Datentyp zu definieren, besteht in der Angabe von *Feldnamen*:

```
> data Tree' a = Leaf' { label :: a }
>              | Branch' { label :: a
>                        , left  :: Tree' a
>                        , right :: Tree' a }
```

Mit `label`, `left` und `right` sind nun *Selektorfunktionen* definiert, mit denen man direkt auf die entsprechenden Komponenten eines `Tree'`-Ausdrucks zugreifen kann.

Benutzerdefinierte Typen lassen sich auch durch *Typsynonyme* einführen. Damit kann jedem beliebigen Typ ein eigener Bezeichner zugewiesen werden. Das folgende Beispiel tut das für einen *Funktionstyp*, also für einen mit dem entsprechenden *Typkonstruktor* `->` konstruierten Typ:

```
> type TreeMod a = Tree a -> Tree a
```

Wie man sieht, ist auch hier Polymorphie erlaubt. `TreeMod a` kann nun für alle Funktionen als Typ angegeben werden, die einen Baum unter Beibehaltung des Knotentyps modifizieren.

Die Funktion `cut` soll nun einen Baum vom Typ `Tree a` an denjenigen Knoten abschneiden, deren Markierung ein bestimmtes, angegebenes Prädikat erfüllt. Ihre Definition wird eingeleitet durch eine *explizite Typangabe*. Die ist in diesem Fall optional, da der Typ durch den Compiler *inferiert* werden kann, was aber nicht immer möglich ist.

```
> cut :: (a -> Bool) -> TreeMod a
```

Man erkennt, daß es sich um eine *Funktion höherer Ordnung* handelt: Das Prädikat, das im Argument der Funktion gegeben ist, ist selbst eine Funktion (`Bool` ist der vordefinierte Datentyp für Boolesche Werte). Außerdem kann `cut` so gesehen werden, daß es ein Argument akzeptiert und eine Funktion vom Typ `TreeMod` zurückgibt — oder aber so, daß es zwei Argumente (ein Prädikat und einen Baum) akzeptiert und einen Baum zurückgibt. Es sind zwei Sichtweisen desselben Sachverhaltes. Man kann sogar jede Funktion so betrachten, als bräuchte sie nur ein Argument; dieses Phänomen nennt man *currying*.

Die Funktionsdefinition selbst geschieht hier zweckmäßigerweise mit direkter Angabe beider Argumente:

```
> cut pred l@(Leaf _) = l

> cut pred (Branch x l r)
>   | pred x    = Leaf x
>   | otherwise = Branch x new_l new_r
>   where
>   new_l = cut pred l
>   new_r = cut pred r
```

Eine ganze Reihe von Programmierstilmitteln wird hier benutzt:

Pattern Matching: `cut` unterscheidet nach dem Konstruktor, mit dem sein zweites Argument aufgebaut ist (`Leaf` oder `Branch`). Die Unterscheidung erfolgt durch die explizite Angabe des Konstruktors. Solche Patterns sind an vielen Stellen der Syntax von Haskell anzutreffen; hier ist es eben in einer Funktionsdefinition.

Guards: Im `Branch`-Fall wird weiter unterschieden danach, ob bestimmte Bedingungen zutreffen (den Wert `True` haben). Dies geschieht durch Angabe eines `|` vor der Bedingung. `otherwise` fängt jeden übriggebliebenen Fall ab, da es als `True` definiert ist.

As-Patterns: Das Zeichen `@` in `Leaf`-Fall dient dazu, das `Leaf`-Konstrukt zusätzlich an den Variablennamen `l` zu binden, damit es auf der rechten Seite der Definition nicht extra über `Leaf...` referenziert werden muß.

Wildcard-Pattern: Das Zeichen `_` in der obersten Zeile steht für einen beliebigen, nicht weiter verwendeten Ausdruck. Wenn schon bekannt ist, daß ein `Leaf`-Ausdruck vorliegt, ist ja die Marke des Blattes unwichtig.

Lokale Definitionen: Ein `where` leitet lokale Funktionsdefinitionen ein. Die Funktionen `new_l` und `new_r` sind nur innerhalb der Definition von `cut` bekannt.

Klammernvermeidung durch Layout: Die letzten zwei Zeilen sind lokale Definitionen. Woher weiß aber ein Compiler, daß `new_r` nicht schon wieder eine *globale* Definition so wie `cut` ist? Es wird von den Layout-Regeln von Haskell erkannt, da die Definition von `new_r` in der gleichen Spalte beginnt wie die von `new_l`. Dies fördert doppelt die Übersicht: einerseits durch das Layout selbst und andererseits durch die damit erreichte Einsparung großer Mengen von Trennzeichen und Klammern.

Kapitel 3

Die Kernsprache Λ

In diesem Kapitel geht es um die funktionale Kernsprache, die in der Kontextanalyse benutzt wird. Bei der Vorstellung der Sprache und der Herleitung ihrer Eigenschaften halten wir uns weitgehend an [Sch99b] und [Man99]. Die erstere der genannten Quellen befand sich während der Anfertigung dieser Arbeit in stetiger Bearbeitung, so daß hier eine Mischung aus Sätzen der verschiedenen Versionen von [Sch99b] sowie von [Man99] entstanden ist. Es entstehen aber keine Konflikte, da das Kapitel eine eigene Konsistenz aufweist.

Eine Kernsprache zeichnet sich dadurch aus — wie der Name bereits andeutet —, daß ihre Syntax gegenüber gewöhnlichen funktionalen Sprachen stark vereinfacht ist, sie aber dennoch mächtig genug ist, jedes funktionale Programm auszudrücken. Jedes Programm einer funktionalen Sprache kann also unter Beibehaltung der Semantik in ein Programm der Kernsprache transformiert werden.

So stellt das hier benutzte Λ die wesentlichen Elemente funktionaler Sprachen zur Verfügung: Funktionsdefinitionen und -anwendungen. Weiterhin verwendet sie algebraische Datentypen und Pattern Matching, wie es in vielen funktionalen Sprachen der Fall ist. Die Auswertung ist nichtstrikt.

3.1 Syntax

Die Kernsprache Λ ist eine vereinfachte Fassung der in [PJL91] benutzten. Gegenüber dort fehlen `let`-Ausdrücke und λ-Funktionsabstraktionen. Diese Konstrukte sind aber durch *Lambda-Lifting* in Superkombinatoren übersetzbar ([PJ87] bzw. [PvE93]), so daß die Ausdrucksfähigkeit der Sprache gewahrt bleibt.

Die Syntax von Programmen und Ausdrücken der Kernsprache Λ wird jeweils durch eine Grammatik definiert, deren Produktionsregeln P Abbildung 3.1 angibt. Variablen- (N) und Terminalsymbole (T) der Grammatiken gehen dort implizit durch Kursiv- bzw. Schreibmaschinenschrift hervor.

Definition 3.1 (Kernsprache Λ). *Die funktionale Kernsprache Λ ist die von der Grammatik $G = (N, T, R, expr)$ erzeugte Sprache. Die Elemente von Λ werden Λ-*Ausdrücke *oder Λ-*Terme *genannt.*

*Ein Λ-*Programm *ist ein Element der Sprache, die von der Grammatik $P = (N, T, R, prog)$ erzeugt wird.*

Ein Kernsprachenprogramm besteht somit aus einer Reihe von Definitionen; diese definieren Funktionskonstanten, die sogenannten *Superkombinatoren*. Ein Superkombinator f heißt n-stellig, geschrieben $\alpha(f) = n$, wenn seine Definition genau n Variablen auf der linken Seite erwähnt.

„Gesättigte Konstruktoren“ sind die Datenobjekte der Sprache: $c \equiv$ `Cons{`t`,`a`}` [1] ist ein a-

[1] Das Symbol $\equiv$ steht in der gesamten Arbeit, wo nicht anders eklärt, für die syntaktische Gleichheit zweier Ausdrücke.

Programm	*program*	→	sc_1 ; ... ; sc_n [;]	$n \geq 1$
Superkombinator	*sc*	→	*var* $var_1 \ldots var_n$ = *expr*	$n \geq 0$
Ausdruck	*expr*	→	*expr aexpr*	Anwendung
		|	$expr_1$ *binop* $expr_2$	Binäre Infix-Anwendung
		|	`case` *expr* `of` *alts*	Case-Ausdruck
		|	*aexpr*	Atomarer Ausdruck
Atomarer Ausdruck	*aexpr*	→	*var*	Variable
		|	*num*	Zahl
		|	`Cons{`*num*`,`*num*`}`	Konstruktor
		|	`(` *expr* `)`	Geklammerter Ausdruck
Alternativen	*alts*	→	alt_1 ; ... ; alt_n	$n \geq 1$
	alt	→	`<`*num*`>` $var_1 \ldots var_n$`->` *expr*	$n \geq 0$
Binärer Operator	*binop*	→	*arithop* | *relop* | *boolop*	
	arithop	→	`+` | `-` | `*` | `/`	Arithmetisch
	relop	→	`<` | `<=` | `==` | `~=` | `>=` | `>`	Vergleich
	boolop	→	`&` | `|`	Boolesch
Variable	*var*	→	*alpha* $varch_1 \ldots varch_n$	$n \geq 0$
	alpha	→	`a` | ... | `z`	
	varch	→	*alpha* | *ziffer* | `_`	
Zahl	*num*	→	$ziffer_1 \ldots ziffer_n$	$n \geq 1$
	ziffer	→	`0` | ... | `9`	

Abbildung 3.1: Syntax der Kernsprache

stelliger Konstruktor, geschrieben $\alpha(c) = a$; seine Anwendung auf a Argumente bildet ein Datenobjekt. t dient zur Unterscheidung verschiedener Konstruktoren; siehe dazu auch Abschnitt 3.1.1.

Wir werden in dieser Arbeit mitunter n-stellige Tupelkonstruktoren `(,...,)` benutzen, so daß deren Anwendung $(a_1, \ldots, a_n)$ geschrieben werden kann. Auch benutzen wir die Schreibweise a`:`b bzw. `[]` für die Konstrukte `Cons{2,2}` a b bzw. `Cons{1,0}` des Datentyps *List*.

Zu beachten ist, daß Λ-Ausdrücke im allgemeinen nicht für sich selbst stehen, da sie Superkombinatoren enthalten können. Sie müssen also im Bezug zu einem Programm gesehen werden, das jene definiert. Wenn im folgenden von Superkombinatoren oder Definitionen die Rede ist, so wird stillschweigend davon ausgegangen, daß ein globales Programm gegeben ist, welches alle nötigen Definitionen aufführt.

Wir gehen im weiteren davon aus, daß jedes Programm eine Definition `bot = bot` enthält. Die Definitionen aller weiteren in dieser Arbeit verwendeten Superkombinatoren sind, sofern nicht an Ort und Stelle gegeben, in Anhang B zu finden.

3.1.1 Typsystem und alternatives `case`

Es soll für jedes Programm eine Menge $\mathcal{A}$ von algebraischen Datentypen $A_1, \ldots, A_n$ geben. Ein Datentyp A wird durch eine Anzahl $|A|$ von Konstruktoren $\mathtt{c}_{A,i}$, $1 \leq i \leq |A|$ spezifiziert. Jeder Konstruktor $\mathtt{c}_{A,i}$ entspricht einem Kernsprachenkonstruktor `Cons{`$t_{A,i}$`,`$\alpha(\mathtt{c}_{A,i})$`}`.

Eine Unterscheidung zwischen verschiedenen Datentypen ist in der Kernsprache allerdings nicht

möglich. Wir werden deshalb in manchen theoretischen Betrachtungen eine alternative Syntax für `case`-Ausdrücke benutzen, die Datentypen verwendet:

Für jeden Typ A soll Λ eine Konstante $\texttt{case}_A$ enthalten:

$$expr \to \texttt{case}_A$$

ist die zugehörige Produktionsregel für die Syntax.

Die Semantik dieser Terme wird weiter unten zusammen mit den anderen `case`-Ausdrücken behandelt. Damit beide Konstrukte gleichwertig werden, also dieselben Fallunterscheidungen tätigen, fordern wir, daß jeder `case`-Ausdruck der zuerst eingeführten Form *vollständig spezifiziert* ist, also für jeden Konstruktor des jeweiligen Datentyps eine Alternative aufführt. Dann nämlich sind beide Formen untereinander austauschbar, wie in [Man99] bewiesen wird (die Konvertierung muß dabei neue Superkombinatoren einführen). Damit sind sie nach Belieben zu verwenden.

3.2 Operationale Semantik

Ein Ausdruck wird ausgewertet, indem man ihn *reduziert* und das Ergebnis beobachtet.

Wir definieren zunächst einige Konstrukte und Bezeichnungen, die in diesem Abschnitt erstmalig benutzt werden:

Definition 3.2 (Teilausdruck). *Sei t ein Ausdruck einer Sprache M, die über eine Grammatik definiert ist. τ sei der Syntaxbaum von t. Ein Ausdruck $t' \in M$ mit Syntaxbaum τ' ist genau dann ein* Unter- *oder* Teilausdruck *von t, wenn τ' ein Teilbaum von τ ist.*

Definition 3.3 (Syntaktischer Kontext). *Ein* syntaktischer Kontext *oder* Termkontext $C[\cdot]$ *über einer Sprache ist ein Ausdruck dieser Sprache (das wird in dieser Arbeit immer Λ sein), der mit einem „Loch" versehen ist: An einer Stelle des Syntaxbaumes, die einem Unterausdruck entspricht, befindet sich anstatt dessen ein „Loch" (·). (Da es um den Syntax*baum *und nicht um den Graphen geht, taucht das Loch tatsächlich nur einmal auf im Ausdruck.) Durch* Einsetzen *eines Ausdrucks a der Sprache in die Position des Lochs, geschrieben $C[a]$, entsteht ein echter Ausdruck der Sprache.*

Sofern der Zusammenhang keine Verwechslung zuläßt, schreiben wir für einen Termkontext auch einfach *Kontext.*

Definition 3.4 (Freie und gebundene Vorkommen einer Variablen). *Eine Variable x innerhalb eines Λ-Ausdrucks t kommt an einer Stelle von t* gebunden *vor, wenn sie dort innerhalb einer* `case`*-Alternative* `<i>` $x_1 \ldots x_m$`->` b_i *steht, so daß $x_j \equiv x$ für ein j.*

Alle anderen Vorkommen von x in t heißen frei.

Definition 3.5 ($t[e_1/x_1, \ldots, e_n/x_n]$). *Sei $t \in \Lambda$. Der Λ-Ausdruck $t[e_1/x_1, \ldots, e_n/x_n]$ entsteht aus t dadurch, daß für $i = 1, \ldots, n$ jedes freie Vorkommen von x_i in t durch e_i ersetzt wird.*

3.2.1 Reduktion

Die $\to_B$-Reduktion ist der Kernmechanismus der Auswertung von Termen. Eine Anwendung eines n-stelligen Superkombinators auf n Argumente und ebenso eine Anwendung eines `case` auf ein Datenobjekt stellt einen *Redex* dar und kann direkt reduziert werden:

Definition 3.6 ($\to_B$).

$$\texttt{f}\ e_1\ \ldots\ e_n \to_B t[e_1/x_1, \ldots, e_n/x_n],$$

wenn `f` *definiert ist durch* `f` $x_1\ \ldots\ x_n$ `=` t.

$$\texttt{case Cons\{}t\texttt{,}n\texttt{\}}\ e_1\ \ldots\ e_n\ \texttt{of}\ a_1\texttt{;}\ldots\texttt{;}a_m \to_B b_j[e_1/x_1, \ldots, e_n/x_n],$$

wenn a_j die Form `<t>` $x_1 \ldots x_n$`->` b_j *hat; und für die andere* `case`*-Form:*

$$\texttt{case}_A\ (\texttt{c}_{A,i}\ e_1\ \ldots\ e_{\alpha(\texttt{c}_{A,i})})\ b_1\ \ldots\ b_{|A|} \to_B b_i\ e_1\ \ldots\ e_{\alpha(\texttt{c}_{A,i})}.$$

Diese Reduktion kann nun erweitert werden auf Ausdrücke, die Redexe enthalten:

Definition 3.7 ($\rightarrow_\delta$). *Die* δ-Reduktion $\rightarrow_\delta$ *ist die kleinste Relation, bei der für beliebige Kontexte* $C[\cdot]$ *gilt:*

$$s \rightarrow_B t \implies C[s] \rightarrow_\delta C[t] \tag{3.1}$$

Um schließlich bei mehreren Möglichkeiten, einen Ausdruck zu δ-reduzieren, eine einzige auszuwählen, geben wir die Normalordnungsreduktion $\rightarrow_{no}$ an. Die Auswahl geschieht durch Einschränkung der Kontexte, in denen die Redexe auftauchen können, auf *Reduktionskontexte*:

Definition 3.8 (Reduktionskontext). *Ein* Reduktionskontext *ist ein Kontext* $R[\cdot]$ *von einer Form, wie sie die folgende Grammatik angibt:*

$$\begin{array}{lcl} R[\cdot] & \rightarrow & \cdot \\ & | & R[\cdot]\ e \\ & | & \texttt{case}\ R[\cdot]\ \texttt{of}\ a_1\texttt{;}\ldots\texttt{;}a_m \end{array}$$

Definition 3.9 ($\rightarrow_{no}$). *Eine Reduktion* $s \rightarrow_\delta t$ *ist in* Normalordnung*, oder kurz:* s no-reduziert *zu* t*, geschrieben* $s \rightarrow_{no} t$*, wenn* (3.1) *für einen Reduktionskontext* $R[\cdot] = C[\cdot]$ *gilt. Der Redex, der durch den Reduktionskontext angegeben ist, wird als* no-Redex *bezeichnet.*

Der Einfachheit halber werden wir die Begriffe δ- und no-Reduktion auch jeweils für die reflexiv-transitive Hülle der eigentlichen Reduktion benutzen. In der mathematischen Notation drücken wir diese Erweiterung durch einen Stern aus: $\stackrel{*}{\rightarrow}_\delta$, $\stackrel{*}{\rightarrow}_{no}$.

3.2.2 Abstraktion von Λ

Jede in einem Λ-Ausdruck auftretende Variable ist gewöhnlich durch ein `case` gebunden. Ausdrücke, die diese Bedingung erfüllen, nennt man *geschlossen*.

Die Kontextanalyse benutzt aber freie Variablen enthaltende, sogenannte *offene* oder *abstrakte* Terme, die ja auch in Λ enthalten sind, und macht darüber Aussagen, welche konkreten (also geschlossenen) Terme diese Variablen unter gegebenen Bedingungen annehmen können. Mit den offenen soll möglichst so wie mit allen anderen Λ-Termen umgegangen werden; gerade die Reduktion soll sich nicht am Vorhandensein freier Variablen stören.

Es ist festzustellen, daß Reduktionen mit freien Variablen keine Probleme bereiten: Die Definitionen aus dem vorangegangenen Abschnitt sind ohne weiteres auch für abstrakte Terme zu gebrauchen. In diesem Fall spricht man von *abstrakter Reduktion*.

Freie Variablen müssen damit nicht gesondert behandelt werden. Auch in den nächsten Abschnitten wird es nur minimal nötig sein, auf die Möglichkeit offener Ausdrücke einzugehen.

3.2.3 Die schwache Kopfnormalform

Als Ergebnis einer Kette von no-Reduktionen erhalten wir (hoffentlich; Ausnahmen werden im nächsten Abschnitt beschrieben) einen Term, der eine schwache Kopfnormalform oder WHNF (englisch „weak head normal form“) ist:

Definition 3.10 (WHNF). *Ein Ausdruck steht in* WHNF*, wenn er*

1. *die Anwendung eines* n*-stelligen Konstruktors auf höchstens* n *Argumente ist („nicht-übersättigter Konstruktor“),*
2. *die Anwendung eines* n*-stelligen Superkombinators auf weniger als* n *Argumente ist („ungesättigter Superkombinator“) oder*
3. *die Form* $R[x]$ *hat für einen Reduktionskontext* R *und eine Variable* x*.*

Ein solcher Ausdruck kann nicht weiter no-reduziert werden.
Man kann WHNFs nach Kategorien unterscheiden:

- CWHNFs (englisch „constructor WHNF“) sind WHNFs aus Punkt 1 der obigen Definition;
- SCWHNFs (englisch „saturated constructor WHNF“) sind gesättigte Konstruktoren;
- FWHNFs (englisch „function WHNF“) sind ungesättigte Konstruktoren oder Superkombinatoren;
- OWHNFs (offene WHNFs, englisch „open WHNF“) sind WHNFs aus Punkt 3 der Definition.

Dabei läßt sich offenbar jede WHNF in genau eine der letzten drei Kategorien einordnen. Der letzte Punkt ist nur für abstrakte Ausdrücke nötig.

Wir sagen, ein Term t *hat* eine WHNF bzw. F-, C-, SC- oder OWHNF, wenn es von t ausgehend eine Folge von δ-Reduktionen gibt, so daß der entstehende Term in der jeweiligen Form steht. In [Sch99b] wird bewiesen, daß man aufgrund einer Standardisierungsmöglichkeit von δ-Reduktionen diese Eigenschaft äquivalent mit no-Reduktionen formulieren kann: Ein Term t hat eine WHNF bzw. F-, C-, SC- oder OWHNF, wenn er im Zuge der Normalordnungsreduktion irgendwann zu einem Term der jeweiligen Form wird.

Zum Beispiel gilt für `bot`:

Lemma 3.11. `bot` *hat keine WHNF.*

Beweis. Es gilt $\texttt{bot} \to_{no} \texttt{bot}$. Die fortgesetzte no-Reduktion bildet also eine Endlosschleife; und `bot` selbst ist keine WHNF. ◇

Wir schreiben im Zusammenhang mit WHNFs:

$s\Uparrow$	wenn s keine WHNF hat,
$s\Downarrow$	wenn s eine WHNF hat,
$s\Downarrow t$	wenn s die WHNF t hat,
$s\Downarrow_S$	wenn s eine SCHWHNF hat,
$s\Downarrow_S t$	wenn s die SCHWHNF t hat,
$s\Downarrow_F$	wenn s eine FWHNF hat.

Man spricht von der *Konvergenz* bzw. *Divergenz* von s, wenn $s\Downarrow$ bzw. $s\Uparrow$.

Damit lassen sich die *primitiven Λ-Terme* definieren, wie sie später bei den Kontexten gebraucht werden:

Definition 3.12 (Λ_p). *Die Menge Λ_p der* primitiven Λ-Terme *ist die kleinste Menge, die*

- *alle divergierenden Terme,*
- *alle Terme mit FWHNF und*
- *alle Terme mit SCWHNF* `c` $a_1 \ldots a_n$ *und primitiven* $a_1, \ldots, a_n$

enthält.

3.3 Wohlgetyptheit

Nicht alle syntaktisch korrekten Programme sind auch semantisch sinnvoll. Beispielsweise erlaubt es die Syntax, Datenkonstruktoren auf eine über ihre Stelligkeit hinausgehende Anzahl von Argumenten anzuwenden. Diesen sogenannten *übersättigten Konstruktoren* kann keine „vernünftige“ Bedeutung zugemessen werden.

Auch ist es möglich, das (alternative!) `case`-Konstrukt eines bestimmten Typs mit einem Term zu benutzen, der eine SCWHNF eines anderen Typs hat oder eine FWHNF. Solche Terme werden ebenfalls als unzulässig angesehen.

Wir werden nun erklären, welche Programme *wohlgetypt* sind. Im Hintergrund steht hierbei der Gedanke, daß Anforderungskontexte alle nicht-wohlgetypten Terme gleich behandeln sollten. Die beiden eben erwähnten Arten von Fehlern bilden dabei den Kern der Definition; die dort angesprochenen Ausdrücke werden naheliegenderweise als *direkt nicht wohlgetypt* bezeichnet.

Definition 3.13 (DNWT). *Ein Λ-Term ist* direkt nicht wohlgetypt, *wenn er eine der folgenden Formen hat:*

- $\mathtt{case}_A\ (\mathtt{c}_{A',i}\ e_1\ \ldots\ e_{\alpha(\mathtt{c}_{A',i})})\ a_1\ \ldots\ a_{|A|}$ *mit* $A \neq A'$,
- $\mathtt{case}_A\ a\ b_1\ \ldots\ b_{|A|}$ *und* a *ist in FWHNF,*
- $\mathtt{c}_{A,i}\ e_1\ \ldots\ e_n$ *mit* $n \geq \alpha(\mathtt{c}_{A',i})$.

Die Menge aller dieser Ausdrücke heißt DNWT.

Allgemein nicht wohlgetypt sind diejenigen Terme, deren Normalordnungsreduktion auf Ausdrücke stößt, bei denen ein Term aus DNWT reduziert werden müßte.

Definition 3.14 (NWT).

$$NWT := \{\, a \in \Lambda \mid \exists b : a \xrightarrow{*}_{no} b \wedge b \equiv R[t] \wedge t \in DNWT \wedge R[\cdot] \text{ ist Reduktionskontext } \}.$$

Alle anderen Ausdrücke sind schließlich wohlgetypt.

Definition 3.15 (WT).

$$WT := \Lambda \setminus NWT.$$

Wir beschließen diesen Abschnitt mit einigen Beobachtungen:

Lemma 3.16. *1.* $\mathtt{bot}$ *ist wohlgetypt.*

2. Alle konvergierenden Terme sind wohlgetypt.

3. Nicht-wohlgetypte Terme können nicht zu einem wohlgetypten Term reduzieren.

4. Alle Terme in NWT divergieren.

3.4 Kontextuelle Ordnung

Wir werden in diesem Abschnitt eine Relation einführen, die die Elemente von Λ bezüglich ihres Informationsgehaltes ordnet. Dadurch wird es möglich sein, einen Λ-Term durch andere zu approximieren. Zum Beispiel wird man unendliche Datenstrukturen durch eine Folge endlicher darstellen können. Auch der Begriff der Äquivalenz zweier Ausdrücke wird erst in diesem Abschnitt erklärt werden. Beide Konzepte werden später bei den Anforderungskontexten gebraucht, um die Menge der Terme zu definieren, die sie repräsentieren.

Zunächst benötigen wir ein wenig Vorbereitung:

Definition 3.17 (Präordnung). *Eine* Präordnung *auf einer Menge M ist eine reflexive und transitive Relation $\leq_M \subseteq M \times M$, kurz auch $\leq$ geschrieben. Der symmetrische Kern dieser Relation wird $\equiv_M$ genannt.*

Definition 3.18 (Kette). *Eine M-*Kette *oder kurz* Kette *ist eine unendliche Folge $s_1 \leq s_2 \leq \ldots$ von Elementen einer Menge M, auf der eine Präordnung $\leq$ definiert ist. Sie wird auch als* aufsteigende Kette *bezeichnet.*

Definition 3.19 ($\bigsqcup$). *Ein Element s einer Menge M, auf der eine Präordnung $\leq$ definiert ist, wird* kleinste obere Schranke *oder* lub *(englisch „least upper bound“) einer Kette s_i genannt, geschrieben $s = \bigsqcup_{M\,i} s_i$ oder kurz $\bigsqcup_i s_i$, wenn*

1. $\forall i : s_i \leq s$ *und*
2. $\forall t : [(\forall i : s_i \leq t) \implies s \leq t]$.

Die Ordnung[2] auf Λ wird über das Verhalten in allen möglichen Termkontexten definiert:

Definition 3.20 ($\leq_c$). *Seien* $s, t \in \Lambda$.

$$s \leq_c t \;:\Longleftrightarrow\; \forall C[\cdot] : C[s]{\Downarrow} \implies C[t]{\Downarrow}$$
$$s \equiv_c t \;:\Longleftrightarrow\; s \leq_c t \wedge t \leq_c s$$

Wir sagen, s *ist* kontextuell kleiner oder gleich t *bzw.* äquivalent *zu* t.

Lemma 3.21. $\leq_c$ *ist eine Präordnung auf* Λ.

Beweis. Die Reflexivität und Transitivität von $\leq_c$ folgen direkt aus den gleichen Eigenschaften von $\implies$. ◇

Mit dieser Definition der Ordnung auf Λ finden sich alle nicht-wohlgetypten Ausdrücke zusammen mit den Termen ohne WHNF am unteren Ende der Hierarchie.

Lemma 3.22.

$$(a \in NWT \vee a{\Uparrow}) \iff \forall s \in \Lambda : a \leq_c s$$

Beweis.
- „$\implies$“: Sei $a \in \text{NWT} \vee a{\Uparrow}$. Wenn nun für einen Kontext $C[\cdot]$ gilt: $C[a]{\Downarrow}$, muß für alle $s \in \Lambda$ auch $C[s]$ eine WHNF haben. Der Grund dafür ist, daß in der Normalordnungs-Reduktionsfolge von $C[a]$ niemals versucht wird, a zu reduzieren, denn das würde entweder zu einer endlosen Reduktion führen (wenn a wohlgetypt ist) oder Nichtwohlgetyptheit von $C[a]$ bedeuten (wenn a nicht wohlgetypt ist). Beides stünde im Widerspruch zu $C[a]{\Downarrow}$. Alle Reduktionen finden also außerhalb des Lochs von $C[\cdot]$ statt und sind so unabhängig vom eingesetzten Term s.
- „$\Longleftarrow$“: Es gelte $\forall s \in \Lambda : a \leq_c s$. Insbesondere gilt $a \leq_c \texttt{bot}$. Aus der Definition von $\leq_c$ sieht man nun mit dem speziellen Kontext $C[\cdot] = \cdot$, daß die Annahme $a{\Downarrow}$ zum Widerspruch $\texttt{bot}{\Downarrow}$ führen würde. Also gilt $a{\Uparrow}$.

◇

Als Repräsentant der Terme, die kontextuell kleiner oder gleich allen Λ-Termen sind, werden wir häufig `bot` benutzen.

Korollar 3.23.

$$\forall s \in \Lambda : \texttt{bot} \leq_c s$$

Beweis. `bot` hat keine WHNF, daher gilt $\texttt{bot}{\Uparrow}$ und das vorige Lemma ist anwendbar. ◇

Kontextuelle Äquivalenz bedeutet gleiches Auswertungsverhalten in allen Kontexten. Dies ist zum Beispiel gegeben, wenn ein Ausdruck auf einen anderen no-reduziert, oder wenn ein Teilausdruck durch einen äquivalenten Teilausdruck ersetzt wird.

Lemma 3.24. *Seien* $s, t \in \Lambda$ *und* $C[\cdot]$ *ein Kontext über* Λ. *Es gelten:*

1. $s \xrightarrow{*}_{no} t \implies s \equiv_c t$,
2. $s \equiv_c t \implies C[s] \equiv_c C[t]$.

Beweis. 1. $s \equiv_c t$ ist über die Konvergenz von s und t definiert. Diese ist ihrerseits mittels der Normalordnungsreduktion erklärt. Wenn nun s zu t no-reduziert, ist daher das Konvergenzverhalten der beiden Ausdrücke gleich. Das gleiche gilt für $C[s]$ und $C[t]$, da entweder in beiden Ausdrücken im Loch von $C[\cdot]$ reduziert wird oder in keinem.

[2] Der Begriff „Ordnung“ wird hier salopp auch für Präordnungen verwendet.

2. Dies folgt sofort aus Definition 3.20, da jeder Kontext, in den $C[s]$ oder $C[t]$ eingesetzt wird, auch ein Kontext um s bzw. t ist.

◇

Bemerkung. *Kontextuelle Äquivalenz gilt auch für Ausdrücke, die durch δ-Reduktion ineinander übergehen. Das gründet sich auf die Konfluenz von Λ, siehe dazu [Sch99b].*

Eine wichtige und einfache Beobachtung ist, daß Funktionsanwendungen die $\leq_c$-Struktur respektieren:

Definition 3.25 (Monotonie). *Sei $f \in \Lambda$. f heißt* monoton *(bezüglich $\leq_c$), wenn*

$$\forall s,t \in \Lambda : s \leq_c t \implies f\ s \leq_c f\ t.$$

Lemma 3.26. *Alle Λ-Ausdrücke sind monoton.*

Beweis. Für $f,s,t \in \Lambda$ folgt $f\ s \leq_c f\ t$ sofort aus $s \leq_c t$, wenn man in der Definition von $\leq_c$ nur Kontexte $C[\cdot] = C'[f\ \cdot]$ betrachtet. ◇

Nun, da wir mit der Ordnungsrelation eine Möglichkeit haben, Ausdrücke bezüglich ihres Informationsgehaltes zu vergleichen, können wir darangehen, Approximationen von Λ-Termen zu bilden. Für unendliche Folgen spielt dabei ein eingeschränkter Begriff der kleinsten oberen Schranke eine Rolle:

Definition 3.27 ($\bigsqcup^c$). *Sei $s_1, s_2, \ldots$ eine Λ-Kette. Ein Ausdruck s heißt* kontextuell kleinste obere Schranke *oder* club *der s_i, geschrieben $s \equiv_c \bigsqcup_i^c s_i$, genau dann, wenn folgende Bedingungen erfüllt sind:*

$$s \equiv_c \bigsqcup_i s_i \tag{3.2}$$

und

$$\forall C[\cdot] : (\forall i : C[s_i]{\Uparrow}) \implies C[s]{\Uparrow} \tag{3.3}$$

Die Eigenschaft von s, club der s_i zu sein, gilt auch für alle zu s äquivalenten Terme, ist aber ansonsten eindeutig:

Lemma 3.28. *Sei $s \equiv_c \bigsqcup_i^c s_i$. Dann gilt*

$$t \equiv_c s \iff t \equiv_c \bigsqcup_i^c s_i.$$

Beweis.
- „$\Longrightarrow$“: (3.2) ist offensichtlich durch $t \equiv_c s \equiv_c \bigsqcup_i s_i$ gegeben. Auch Bedingung (3.3) gilt für t, da $C[s]{\Uparrow}$ und $C[t]{\Uparrow}$ logisch äquivalent sind, wie aus Definition 3.20 ersichtlich ist.
- „$\Longleftarrow$“: Aus (3.2) ergibt sich mit der Transitivität von $\equiv_c$ sofort $t \equiv_c s$.

◇

Es zeigt sich, daß Funktionsanwendungen auch die club-Bildung respektieren. Dazu führen wir das zentrale Konzept der *Stetigkeit* ein:

Definition 3.29 (Stetigkeit). *Ein Λ-Ausdruck f heißt* stetig *(bezüglich $\bigsqcup^c$), wenn für alle Ketten $s_1, s_2, \ldots$ von Λ-Ausdrücken, deren club $s \equiv_c \bigsqcup_i^c s_i$ existiert, gilt:*

$$f\ s \equiv_c \bigsqcup_i^c (f\ s_i).$$

Die obige Definition besteht im wesentlichen aus zwei Teilen: Einmal, daß $\bigsqcup_i^c (f\ s_i)$ überhaupt existiert (denn das ist nicht selbstverständlich), und zusätzlich, daß er äquivalent zu $f\ s$ ist.

Eine stetige Funktion besitzt also die Eigenschaft, daß der Funktionswert auf einem Ausdruck s, welcher durch $s_1, s_2, \ldots$ approximiert wird, modulo $\equiv_c$ allein durch die Funktionswerte auf diesen Approximationen bestimmt ist und selbst durch sie approximiert wird.

Lemma 3.30. *Alle Λ-Ausdrücke sind stetig.*

Beweis. Seien f und die Kette s_i mit club s wie in der Definition gegeben. Der Beweis der Stetigkeit von f zerfällt in drei Teile:

1. *$f\ s$ ist obere Schranke von $f\ s_i$:* Dies folgt bereits aus der Monotonie von f, die mit Lemma 3.26 gegeben ist. Denn Voraussetzung ist $\forall i : s_i \leq_c s$; also gilt $\forall i : f\ s_i \leq_c f\ s$.

2. (3.3) *ist für $f\ s_i$ und $f\ s$ erfüllt:* Nach Voraussetzung gilt (3.3) für die s_i und s. Die Behauptung folgt nun leicht, wenn man in (3.3) nur Kontexte $C[\cdot] = C'[f\ \cdot]$ betrachtet.

3. *$f\ s$ ist die* kleinste *obere Schranke von $f\ s_i$:* Sei $t \in \Lambda$ eine obere Schranke der $f\ s_i$, d.h. $\forall i : f\ s_i \leq_c t$. Es ist nun $f\ s \leq_c t$ zu zeigen.

   Für einen beliebigen Kontext $C[\cdot]$ gelte $C[f\ s]{\Downarrow}$. Wie unter dem vorigen Punkt gezeigt, gilt die Beziehung (3.3) für $f\ s_i$ und $f\ s$. Daher muß es ein i geben, für das $C[f\ s_i]{\Downarrow}$. Aus der Voraussetzung $f\ s_i \leq_c t$ folgt damit $C[t]{\Downarrow}$. Da C beliebig zu wählen war, gilt insgesamt $\forall C[\cdot] : C[f\ s]{\Downarrow} \Longrightarrow C[t]{\Downarrow}$, was zu zeigen war.

Damit ist Stetigkeit gegeben. ◇

3.5 club und lub

Der Begriff der kleinsten oberen Schranke formalisiert die Approximation von Λ-Ausdrücken durch andere. Man sieht zum Beispiel leicht, daß ein Ausdruck wie `repeat` x, wobei `repeat` definiert ist durch `repeat a = a : (repeat a)`, durch eine Kette von Datenkonstrukten zunehmender Länge approximiert werden kann, die aus `repeat` x durch wiederholte no-Reduktion und durch einmaliges Ersetzen von `repeat` x durch `bot` hervorgehen, so daß `repeat` x den lub dieser Kette bildet.

Warum überhaupt der stärkere Begriff des clubs eingeführt wurde, ist zunächst nicht klar. Auch die Definition von Stetigkeit ist ohne die Benutzung von clubs denkbar — man könnte dafür lubs benutzen.

Doch liegt hier ein entscheidender Punkt: Die Stetigkeit aller Λ-Ausdrücke, eine wichtige grundlegende Eigenschaft der Kernsprache, konnte bisher ohne die Verwendung der club-Eigenschaft noch nicht bewiesen werden. Die Einschränkung auf clubs als spezielle lubs schließt hier die Lücke, so daß diese Art von Stetigkeit gezeigt werden konnte.

Andererseits wurde auch noch kein Beispiel eines lubs bekannt, der bewiesenermaßen kein club ist. Es ist also möglich, daß jeder lub auch ein club ist. In diesem Fall käme man ganz ohne die Definition eines clubs aus; die entsprechende Eigenschaft könnte bei jedem lub vorausgesetzt werden.

Die Frage, ob es überhaupt nichttriviale clubs gibt, ist allerdings hier auch noch nicht angesprochen worden und genauso berechtigt. In diesem Abschnitt zeigen wir, daß dem so ist und rechtfertigen damit die Einführung der clubs als zumindest nicht abwegig.

Der Satz, der dies zeigt, gebraucht Mehrloch-Kontexte:

Definition 3.31 (Mehrloch-Kontext). *Ein n-stelliger* Mehrloch-Kontext $C[\cdot_1, \ldots, \cdot_n]$ *ist analog zu Definition 3.3 ein mit genau n Löchern versehener Ausdruck.*

Lemma 3.32. *Sei mit* `Y f = f (Y f)` *ein Superkombinator* `Y` $\in \Lambda$ *definiert. Für jeden Λ-Ausdruck f gilt:*

$$\texttt{Y}\ f \equiv_c \bigsqcup_{i \geq 1}^{c} (f^i\ \texttt{bot}).$$

Dabei ist $f^1 = f$ *und* $f^{i+1}\ x = f\ (f^i\ x)$.

Beweis. Der Beweis besteht aus vier Teilen:

1. f^i `bot` *ist eine Kette:* Dies ist eine einfache Induktion: Es ist `bot` $\leq_c f$ `bot` nach Korollar 3.23. Durch die Monotonie von f nach Lemma 3.26 folgen nun alle Bezüge der Kettenelemente.

2. `Y` f *ist obere Schranke von* f^i `bot`*:* Ebenfalls induktiv folgern wir zuerst `bot` $\leq_c$ `Y` f nach Korollar 3.23. Aus der Induktionsannahme f^i `bot` $\leq_c$ `Y` f folgt mit der Monotonie von f: f^{i+1} `bot` $\leq_c f$ (`Y` f). Auf der rechten Seite hätten wir statt f (`Y` f) gerne `Y` f, da dann die Behauptung gezeigt ist. Der eine Ausdruck geht aber durch no-Reduktion aus dem anderen hervor; sie sind also nach Lemma 3.24 äquivalent, was den Beweis schließt.

3. (3.3) *ist für* f^i `bot` *und* `Y` f *erfüllt:*

   $C[\cdot]$ sei ein Kontext, für den C[`Y` f]$\Downarrow$ gilt. Es muß ein i gefunden werden, so daß $C[f^i$ `bot`$]\Downarrow$.

   Dazu versuchen wir, die Reduktion von C[`Y` f] zu WHNF, von der wir ja wissen, daß sie existiert, auf $C[f^i$ `bot`] nachzuvollziehen.

   Auf dem no-Reduktionsweg von C[`Y` f] zur WHNF werden wir nach jedem Schritt einen Mehrloch-Kontext haben, in dessen Löchern nur `Y` f steht. Diese Mehrloch-Kontexte werden übertragen auf den Reduktionsweg von $C[f^i$ `bot`]. Sei nun D_n[`Y` $f, \ldots,$ `Y` f] der Ausdruck nach dem n-ten no-Reduktionsschritt von C[`Y` f]. Terme auf dem Reduktionsweg von $C[f^i$ `bot`] werden dabei die allgemeine Form $D_n[f^{i_{n,1}}$ `bot`$, \ldots, f^{i_{n,k_n}}$ `bot`] haben. Beginnend mit $D_0[\cdot] = C[\cdot]$, unterscheiden wir nun danach, an welcher Stelle die Reduktion stattfindet:

   (a) Es wird nicht im Inneren eines der Löcher reduziert:
   D_n[`Y` $f, \ldots,$ `Y` f] reduziert zu E_n[`Y` $f, \ldots,$ `Y` f].
   E_n hat hierbei eventuell mehr Löcher als D_n: Die Löcher von E_n setzen sich zusammen aus den Löchern von D_n, die sich nicht im no-Redex von D_n befinden, und zusätzlich den Kopien der Löcher innerhalb des no-Redexes von D_n, wie sie durch die Reduktion entstehen.
   Da diese Reduktion nicht vom Inhalt der Löcher abhängig ist, spiegelt sie sich auf der anderen Seite wider in der no-Reduktion $D_n[f^{i_{n,1}}$ `bot`$, \ldots, f^{i_{n,k_n}}$ `bot`$] \rightarrow_{no} E_n[f^{i'_{n,1}}$ `bot`$, \ldots, f^{i'_{n,k'_n}}$ `bot`$]$. Wir setzen $D_{n+1} := E_n$.

   (b) Der Ausdruck `Y` f wird im j-ten Loch reduziert, d.h. D_n[`Y` $f, \ldots,$ `Y` f] $\rightarrow_{no}$ D_n[`Y` $f, \ldots, f$ (`Y` f)$, \ldots,$ `Y` f]:
   Dieser Schritt wird auf $D_n[f^{i_{n,1}}$ `bot`$, \ldots, f^{i_{n,k_n}}$ `bot`] *nicht* durch eine Reduktion nachvollzogen. Statt dessen ändert sich nur der Kontext; es wird sozusagen f aus dem j-ten Loch „herausgezogen“: Wir setzen $D_{n+1}[\cdot, \ldots, \cdot] := D_n[\cdot, \ldots, f\ \cdot, \ldots, \cdot]$, wobei $f\ \cdot$ im j-ten Loch stehe. Somit ist $D_{n+1}[f^{i_{n+1,1}}$ `bot`$, \ldots, f^{i_{n+1,k_{n+1}}}$ `bot`$] = D_n[f^{i_{n,1}}$ `bot`$, \ldots, f^{i_{n,k_n}}$ `bot`] mit $i_{n,j} = i_{n+1,j} + 1$, sofern $i_{n,j} > 1$ ist, denn sonst ist $D_n[f^{i_{n,1}}$ `bot`$, \ldots, f^{i_{n,k_n}}$ `bot`] gar nicht mit D_{n+1} darstellbar. Diese Bedingung $i_{n,j} > 1$ sei aber o.B.d.A. gegeben dadurch, daß i zu Anfang groß genug gewählt wird. Das kann man immer machen, da C[`Y` f] nur endlich viele Reduktionsschritte zur WHNF benötigt.

   Der Reduktionsweg von $C[f^i$ `bot`] besteht also ausschließlich aus solchen Reduktionen, die oben unter die erste Kategorie fallen. Da C[`Y` f] nach endlich vielen Schritten eine WHNF D_n[`Y` $f, \ldots,$ `Y` f] erreicht, reduziert $C[f^i$ `bot`] nach höchstens genauso vielen no-Reduktionsschritten zu $D_n[f^{i_{n,1}}$ `bot`$, \ldots, f^{i_{n,k_n}}$ `bot`]. Es ist leicht zu sehen, daß dieser Term dann ebenfalls eine WHNF haben muß.

4. $f\ s$ *ist die* kleinste *obere Schranke von* $f\ s_i$*:* Dies folgt aus den vorigen beiden Punkten auf die gleiche Weise, wie im Beweis zu Lemma 3.30 vorgegangen wurde. ◇

Das obige Lemma läßt sich nun auf das `repeat`-Beispiel von vorhin anwenden:
`repeat` x ist erkennbar äquivalent zu `Y (`x`:)` und daher club von $(\texttt{(}x\texttt{:)}^i\ \texttt{bot})_i$.

Wir schließen dieses Kapitel mit einem Lemma, das eine Art erweiterter Stetigkeit garantiert, nämlich für mehrere Argumente gleichzeitig. Es wird im Kapitel über Kontexte oft benutzt werden. Man beachte, daß es hier nicht um mehrfach hintereinander gebildete clubs geht (wie z.B. in $\bigsqcup_j^c(\bigsqcup_i^c(f\ x_i\ y_i))$), sondern um eine einfache, „diagonale" club-Bildung ($\bigsqcup_i^c(f\ x_i\ y_i)$).
Dieses Lemma ist weder in [Sch99b] noch in [Man99] aufgeführt. Ein vergleichbares Resultat findet sich aber in [Win93, Prop. 8.9, Lemma 8.10]; dort ist die Sichtweise mehr denotational, und es fehlen club-Betrachtungen.

Lemma 3.33. *Seien* $x_0 \leq_c x_1 \leq_c \ldots$ *und* $y_0 \leq_c y_1 \leq_c \ldots$ *aufsteigende Ketten und* `c` *ein zweistelliger Konstruktor.*

1. `c` $\bigsqcup_i^c x_i\ \bigsqcup_j^c y_j$ *ist club von* `(c` $x_i\ y_i)_{i\geq 0}$.
2. *Für* $k = 1;2$ *ist* $\texttt{proj}_\texttt{c}\ k\ \bigsqcup_i^c(\texttt{c}\ x_i\ y_i)$ *club von* $(x_i)_{i\geq 0}$ $(k=1)$ *bzw.* $(y_i)_{i\geq 0}$ $(k=2)$. *Die Funktion* $\texttt{proj}_\texttt{c}$ *ist hierbei im Pseudocode gegeben durch* $\texttt{proj}_\texttt{c}\ k\ \texttt{(c}\ a_1\ a_2\texttt{)} = a_k$.
3. $\forall a : a \equiv_c \bigsqcup_i^c(\texttt{c}\ x_i\ y_i) \iff a \equiv_c \texttt{c}\ \bigsqcup_i^c x_i\ \bigsqcup_j^c y_j$.
4. $\forall a, f : a \equiv_c \bigsqcup_i^c(f\ x_i\ y_i) \iff a \equiv_c f\ \bigsqcup_i^c x_i\ \bigsqcup_j^c y_j$.
5. $\forall a : a \equiv_c (\bigsqcup_i^c x_i)\ (\bigsqcup_i^c y_i) \iff a \equiv_c \bigsqcup_i^c(x_i\ y_i)$.
6. *Die Punkte 1 bis 4 gelten analog für mehr als zwei Argumente.*

Beweis. 1. O.B.d.A. wird hier der Tupelkonstruktor benutzt.

(a) Für alle i ist $(x_i,y_i) \leq_c (x_i, \bigsqcup_j^c y_j) \leq_c (\bigsqcup_i^c x_i, \bigsqcup_j^c y_j)$; der Ausdruck ist also obere Schranke der Kette.

(b) (3.3) (hier „club-Bedingung" genannt) ist erfüllt für (x_i,y_i) und $(\bigsqcup_i^c x_i, \bigsqcup_j^c y_j)$:
Gelte $C[(\bigsqcup_i^c x_i, \bigsqcup_j^c y_j)]{\Downarrow}$. Es gilt, ein k zu finden, so daß $C[(x_k,y_k)]{\Downarrow}$.
Aus der Annahme folgt mit der club-Bedingung der y_j und mit dem Kontext $D[\cdot] = C[(\bigsqcup_i^c x_i, \cdot)]$, daß es ein j gibt mit $C[(\bigsqcup_i^c x_i, y_j)]{\Downarrow}$. Mit der club-Bedingung der x_i und dem Kontext $E[\cdot] = C[(\cdot\ , y_j)]$ folgt wiederum, daß ein i existiert mit $C[(x_i,y_j)]{\Downarrow}$. Durch die Monotonie aller Ausdrücke und der Ketteneigenschaft der x_i- und y_j-Ketten ergibt sich $C[(x_{max\{i,j\}},y_{max\{i,j\}})]{\Downarrow}$, womit die Behauptung gezeigt ist.

(c) $(\bigsqcup_i^c x_i, \bigsqcup_j^c y_j)$ ist *kleinste* obere Schranke: Das folgt aus den vorangegangenen beiden Punkten genau wie im Beweis von Lemma 3.30.

Nachfolgend wird implizit von Lemma 3.24, Punkt 1 Gebrauch gemacht.

2. Folgt sofort aus der Stetigkeit von $\texttt{proj}_\texttt{c}\ k$.
3. Folgt direkt aus den ersten beiden Punkten.
4. Da $f\ a\ b \equiv_c$ `(uncurry` f`)` (a,b), kann Punkt 3 angewendet werden unter Ausnutzung der Stetigkeit von `uncurry`.
5. Folgt aus Punkt 4 mit $f =$ `apply`.
6. Die Beweise verlaufen analog für mehr Argumente. ◇

Kapitel 4

Kontexte

In diesem Kapitel werden *Anforderungskontexte*, kurz *Kontexte* eingeführt. Sie sind eine Darstellung von Teilmengen von Λ. Zunächst stellen wir die *konventionellen* Kontexte vor. Dabei folgen wir [Sch99b]. In Abschnitt 4.4 definieren wir dann zwei Arten von Funktionskontexten. Im Anschluß an Syntax- und Semantikdefinitionen werden einige für die Kontextanalyse benötigte Sätze bewiesen. Danach folgen Abschnitte über Projektions- und Komplementkontexte, die ebenfalls in der Analyse Anwendung finden.

4.1 Syntax

Der Aufbau eines *konventionellen Kontextes* ist aus der Grammatik in Abbildung 4.1 zu erkennen:

Kontext	E	$\to$	$C \mid S \mid V \mid K \mid N \mid W$	
Konstante	C	$\to$	`Bot` $\mid$ `Fun`	
Schnitt	S	$\to$	$E_1 \cap \ldots \cap E_n$	$n \geq 1$
Vereinigung	V	$\to$	$\langle E_1, \ldots, E_n \rangle$	$n \geq 0$
Konstruktorkontext	K	$\to$	$\mathtt{c}_{A,i}\ E_1 \ldots E_n$	$n \geq 0, A \in \mathcal{A}, 1 \leq i \leq \lvert A \rvert$
Kontextname	N	$\to$	*Kontextname*	
`where`-Ausdruck	W	$\to$	Term(E) `where` $T = N$	
Kontextmuster	T	$\to$	$\mathtt{c}_{A,i}\ T_1 \ldots T_n$	$n \geq 0, A \in \mathcal{A}, 1 \leq i \leq \lvert A \rvert$
		$\mid$	`Bot`	
		$\mid$	X	
Variable	X	$\to$	*Variable*	

Abbildung 4.1: Syntax von Kontexten

Term(E) steht hier für einen Kontext, der freie Variablen enthalten kann, der also über eine analoge Grammatik wie die obige definiert ist mit dem Unterschied, daß die Produktion $E \to X$ zugelassen wird. Ein solcher Ausdruck wird ebenfalls Kontext genannt. Wenn eine Unterscheidung nötig ist, geht sie an gegebener Stelle aus dem Zusammenhang hervor.

Definition 4.1 (Λ^C, $\Lambda^{C\#}$). *Die Menge aller Kontextausdrücke heißt Λ^C.*

Die Menge aller Kontextausdrücke, die freie Variablen enthalten können (also für die Term(E) steht) heißt $\Lambda^{C\#}$.

Vor der Definition der Menge freier Variablen klären wir, was ein Teilkontext ist:

Definition 4.2 (Teilkontexte). *Sei $C \in \Lambda^{C\#}$. Ein* Unter- *oder* Teilkontext *von C ist ein Teilausdruck von C gemäß Definition 3.2. Ein* echter Teilkontext *von C ist ein Teilkontext C' von C mit $C' \not\equiv C$.*

Die Menge $DT(C)$ der direkten Teilkontexte *von C ist wie folgt definiert:*

$$\begin{array}{lcll} DT(\langle C_1,\ldots,C_n\rangle) & := & \{C_1,\ldots,C_n\} & \\ DT(C_1 \cap \ldots \cap C_n) & := & \{C_1,\ldots,C_n\} & \\ DT(\mathtt{c}_{A,i}\ C_1 \ldots C_n) & := & \{C_1,\ldots,C_n\} & \\ DT(S\ \mathtt{where}\ T = N) & := & \{S,T,N\} & \\ DT(C) & := & \emptyset & \textit{sonst} \end{array}$$

Die Menge der freien Variablen $\mathcal{FV}(C)$ eines Kontextes C enthält alle nicht durch ein `where` gebundenen Variablen:

Definition 4.3 ($\mathcal{FV}(C)$).

$$\begin{array}{lcll} \mathcal{FV}(S\ \mathtt{where}\ T = N) & := & \mathcal{FV}(S) \setminus \mathcal{FV}(T) & \\ \mathcal{FV}(V) & := & \{V\} & V\ \textit{Variable} \\ \mathcal{FV}(E) & := & \bigcup\{\,\mathcal{FV}(e) \mid e\ \textit{direkter Teilkontext von}\ E\,\} & \textit{sonst} \end{array}$$

Die hier gegebene Definition von Kontexten stimmt mit der in [Sch99b] überein. Da in Abschnitt 4.4 diese Definition erweitert wird, nennen wir die eben eingeführten Kontexte *konventionell*. Die Bezeichnung Λ^C wird allerdings später auch für die erweiterte Sprache beibehalten. In den wenigen Fällen, in denen wir uns explizit auf konventionelle Kontexte beziehen, wird ein eigenes Symbol, Λ^{C^0}, dafür verwendet werden.

Direkte Teilkontexte eines Kontextes C werden mitunter als *Einträge* oder *Komponenten* von C bezeichnet. Häufig wird das bei Vereinigungen vorkommen. Bei Jecs (s. Abschnitt 4.2) werden diese Teile oft als *Stellen* bezeichnet.

Für Konstruktorkontexte der Listenkonstruktoren werden wir wie in Kapitel 3 `:` in Infixnotation bzw. `[]` schreiben.

Kontextnamen müssen durch Definitionen erklärt sein; eine solche hat die Form $N = E$, N ein Kontextname, E ein Kontext. Wenn wir mit Kontexten zu tun haben, die Namen enthalten, gehen wir stets davon aus, daß sie sich auf eine bestimmte Menge von Kontextdefinitionen beziehen.

Definition 4.4 ($\mathcal{C}, N(\mathcal{C})$). *$\mathcal{C}$ sei eine Menge von Kontextdefinitionen, die dynamisch alle jeweils benutzten Definitionen enthalte. $N(\mathcal{C})$ sei die Menge der in $\mathcal{C}$ definierten Kontextnamen.*

In $\mathcal{C}$ seien immer die folgenden Definitionen enthalten:

$$\begin{array}{l} \mathtt{Fin} = \langle \mathtt{[]}, \mathtt{Top{:}Fin} \rangle \\ \mathtt{Inf} = \langle \mathtt{Bot}, \mathtt{Top{:}Inf} \rangle \\ \mathtt{Top} = \langle \mathtt{Bot}, \mathtt{Fun}, \underbrace{\mathtt{c}_{A,i}\ \mathtt{Top} \ldots \mathtt{Top}, \ldots}_{\substack{\textit{alle Konstruktoren aus } \Lambda, \\ \textit{jeweils gesättigt}}} \rangle \end{array}$$

`Fin` soll dabei endliche Listen repräsentieren. `Inf` steht für unendliche Listen und Listen, deren Ende keine WHNF hat, und `Top` soll für alle Ausdrücke stehen.

Die *primitiven Kontexte* weisen eine besonders einfache Struktur auf:

Definition 4.5 (Λ_p^C). *Die Menge Λ_p^C der* primitiven Kontexte *ist die kleinste Menge, die*

- `Bot`,
- `Fun` *und*
- *alle Konstruktorkontexte* $\mathtt{c}\ a_1 \ldots a_{\alpha(\mathtt{c})}$ *mit primitiven* $a_1, \ldots, a_n$

enthält.

4.2 Joint-evaluation-Kontexte

Die Kontextanalyse kann Anfragen bearbeiten, die mehrere freie Variablen haben. In einer gemeinsamen Lösung für alle diese Variablen sind die Kontexte für die einzelnen Variablen im allgemeinen nicht unabhängig voneinander. Um Beziehungen der Variablen untereinander darstellen zu können, benutzen wir Joint-evaluation-Kontexte, von hier an abgekürzt mit Jecs (englisch *joint evaluation context*: Kontext mit gemeinsamer Auswertung). Das sind Kontexte, die die Werte nicht nur einer, sondern mehrerer Variablen repräsentieren sollen.

Semantisch könnte ihnen an Stelle der üblichen Repräsentation in $\mathcal{P}(\Lambda)$ deswegen eine Teilmenge von $\Lambda \times \cdots \times \Lambda$ zugeordnet werden. Weniger Umstände handelt man sich jedoch ein, indem man Konstruktorkontexte verwendet mit eigens für sie geschaffenen Konstruktoren.

Wir fordern, daß diese Konstruktoren in keinem Λ-Programm vorkommen. Damit gibt es keine Ausdrücke, die zu einer SCWHNF mit einem solchen Konstruktor reduzieren und es selbst nicht sind. Das führt dazu — im Vorgriff auf Abschnitt 4.3 —, daß die Werte, die ein n-stelliger Jec repräsentiert, alle den n-stelligen speziellen Jec-Tupelkonstruktor auf Top-Level besitzen (geschrieben $(x_1, \ldots, x_n)$). Solche Λ-Terme heißen *Jec-Konstrukte*. Man kann also einen Jec in der Tat als Zusammenstellung der Werte der Variablen betrachten, für die der Kontext angefragt wurde.

Einstellige Jecs und Jec-Konstrukte existieren nicht oder seien syntaktisch mit ihrem Inhalt zu identifizieren. Ein nullstelliger Jec sei jedoch definiert: der Jec () besitzt nur den Term () als Konkretisierung.

Jecs sind als Kontexte kein Spezialfall, sondern Konstruktorkontexte mit dem entsprechenden n-stelligen Konstruktor $(, \ldots,)$. Man muß sie also nirgends gesondert behandeln, wenn dort schon Konstruktorkontexte behandelt werden.

Joint-evaluation-Kontexte können prinzipiell auch in Kontexten anderer Form auftreten. Allerdings macht es für uns keinen Sinn, beispielsweise Vereinigungen von Jecs verschiedener Stelligkeit zu betrachten, denn dann verlieren die Jec-Konstruktoren ihre Bedeutung als Zusammenfassung von Werten einer festen Variablenmenge. Semantische Inkonsistenzen dieser Art werden also im allgemeinen nicht auftreten, so daß ein Kontext, dessen Konkretisierung Jec-Konstrukte einer bestimmten Stelligkeit enthält, keine anderen Λ-Terme repräsentiert.

Zur Unterscheidung von den gewöhnlichen Tupeln, die als Typkonstrukte in Λ möglicherweise definiert sind, schreiben wir die Klammern und Kommata in Schreibmaschinenschrift, wenn sie zu einem solchen Ausdruck aus Λ gehören: z.B. `(x,y)`; handelt es sich um einen Joint-evaluation-Kontext, schreiben wir kursiv, wie auch bei dessen repräsentierten Termen mit dem speziellen Jec-Konstruktor: (x, y).

4.3 Semantik

Welche Terme repräsentiert ein Kontext? In diesem Abschnitt wird die Bedeutung der Kontextausdrücke behandelt. Dabei gibt es einerseits die *primitiven Terme* eines Kontextes, deren wesentliche Struktur nur endlich sein kann, und andererseits seine endgültige *Konkretisierung*, die über den ersteren Begriff definiert ist.

4.3.1 Primitive Konkretisierung von Kontexten

In diesem Abschnitt werden die *primitiven* Terme eines Kontextes definiert. Wir beginnen mit den primitiven Kontexten und tasten uns dann mit Hilfe von Fixpunkteigenschaften an die anderen heran.

Primitive Kontexte

Für einen primitiven Kontext T läßt sich die Menge der primitiven Terme $\Lambda_p(T)$, die von ihm repräsentiert werden, einfach induktiv definieren.

Definition 4.6 (primitive Terme der primitiven Kontexte). *Die Menge* $\Lambda_p(T)$ *der* primitiven Terme *ist für* $T \in \Lambda_p^C$ *gegeben durch*

$$\begin{aligned} \Lambda_p(\texttt{Bot}) &:= \{\, t \mid t\Uparrow \,\} \\ \Lambda_p(\texttt{Fun}) &:= \{\, t \mid t\Downarrow_F \,\} \\ \Lambda_p(\texttt{c}\ C_1 \ldots C_n) &:= \{\, t \mid t\Downarrow_S \texttt{c}\ t_1 \ldots t_n \wedge \forall i : t_i \in \Lambda_p(C_i) \,\} \end{aligned}$$

Die folgenden Eigenschaften sind erwähnenswert und werden vor allem in den späteren Abschnitten dieses Kapitels oft benutzt:

Lemma 4.7. *Für jeden* Λ_p*-Ausdruck* t *existiert genau ein primitiver Kontext* C*, so daß* $t \in \Lambda_p(C)$*.*

Lemma 4.8. *Sei* T *ein primitiver Kontext, in dem* `Fun` *nicht vorkommt, und* $x \in \Lambda_p(T)$*.*

$$\Lambda_p(T) = \{\, y \mid y \equiv_c x \,\}$$

Beweis.
- $\subseteq$: Folgt induktiv aus dem Aufbau von T.
- $\supseteq$: Ist klar für $T = \texttt{Bot}$. Andernfalls folgert man $y \in WT$, $y \Downarrow$ und über ein passendes `case`-Konstrukt, daß x und y eine WHNF mit dem gleichen Top-Level-Konstruktor haben. Mit weiteren `case`-Anwendungen kann man den Rest der Struktur von T durchgehen.

◇

Für ein $a \equiv_c \bigsqcup_i^c a_i$ mit $a \in \Lambda_p(P)$ finden sich in seiner approximierenden Kette fast nur Terme aus der gleichen Menge $\Lambda_p(P)$:

Lemma 4.9. *Sei* P *ein primitiver Kontext.*

$$a \in \Lambda_p(P) \wedge a \equiv_c \bigsqcup_i^c a_i \implies \exists i_0 : \forall i > i_0 : a_i \in \Lambda_p(P)$$

Beweis. Für $P = \texttt{Bot}$ ist dies trivial, da dann alle $a_i \equiv_c \texttt{bot}$ sind. Bei $P = \texttt{Fun}$ muß ein a_{i_0} und damit auch alle folgenden a_i FWHNF haben, denn sonst könnte der club nicht kontextuell größer sein als `bot`.

Bei Konstruktorkontexten $P = \texttt{c}\ \ldots$ sieht man mit passend gewählten syntaktischen `case`-Kontexten, daß $a_i \Downarrow_S \texttt{c}\ \ldots$ für alle genügend großen i sein muß. Gleiches folgt für jedes Argument von `c` mit jeweils passenden, eventuell geschachtelten `case`-Kontexten. ◇

Allgemeine Kontexte

Andere Kontextausdrücke enthalten möglicherweise Namen. Diese können rekursiv verwendet sein; die primitiven Λ-Ausdrücke werden deswegen über einen Fixpunkt definiert. Das müssen wir etwas vorbereiten.

Benötigt wird ein Funktionenraum, dessen Elemente Kontextnamen in Teilmengen von Λ abbilden. Darauf wird dann eine Struktur definiert werden, in der kleinste obere Schranken existieren.

Definition 4.10 ($(M \to N)$). *Für zwei Mengen* M *und* N *sei* $(M \to N)$ *die Menge aller Funktionen , die* M *nach* N *abbilden.*

Definition 4.11 ($L, \mathbf{0}$).

$$L := (N(\mathcal{C}) \to \mathcal{P}(\Lambda)) \qquad\qquad \mathbf{0} \in L : x \mapsto \emptyset$$

Definition 4.12 (partielle Ordnung, cpo, cpo$_\perp$). *Eine* partielle Ordnung *auf einer Menge M ist eine reflexive, transitive und antisymmetrische Relation $\leq \subseteq M \times M$.*

Eine vollständige partielle Ordnung *oder* cpo *(englisch: complete partial order) ist eine Menge M mit einer partiellen Ordnung $\leq$ und der folgenden Eigenschaft:*

Jede M-Kette besitzt eine kleinste obere Schranke in M.

Eine cpo$_\perp$ *ist eine cpo M mit kleinstem Element $\perp_M$ oder $\perp$ (gesprochen „bot"), d.h. mit der Eigenschaft: $\exists \perp_M \in M : \forall x \in M : \perp_M \leq x$.*

Zum Beispiel bildet jede Potenzmenge eine cpo$_\perp$ mit $\emptyset$ als dem kleinsten Element, wenn $\leq$ für die Teilmengenbeziehung steht. Die lub-Bildung entspricht hier der Vereinigung.

Lemma 4.13. *L ist eine cpo$_\perp$ mit der punktweisen Ordnung.*

Beweis. $\mathcal{P}(\Lambda)$ ist als Potenzmenge eine cpo$_\perp$. Außerdem ist für alle Mengen S und alle cpo$_\perp$ D auch $(S \to D)$ wieder eine cpo$_\perp$ mit punktweiser Ordnung, nämlich mit $\perp_{(S \to D)} = f : x \mapsto \perp_D$ und $(\sqcup_{(S \to D)})_i f_i = f : x \mapsto \sqcup_{D i} f_i(x)$, denn $f_i(x)$ ist wegen der punktweisen Ordnung eine Kette, falls f_i eine ist, und ein lub für Ketten existiert in D. ◇

Substitutionen behandeln Variablen in Kontextausdrücken:

Definition 4.14 (Σ, Σ_p). *Eine* Substitution *auf Kontexten ist eine Abbildung $\sigma : \Lambda^{C\#} \to \Lambda^{C\#}$, die die Struktur von Kontexten respektiert und die folgenden Eigenschaften besitzt:*

$$\begin{array}{ll} \sigma(C) = C & \text{wenn } C \in \Lambda^C, \\ \sigma(x) = x \vee \sigma(x) \in \Lambda^C & \text{wenn } x \text{ eine Variable ist.} \end{array}$$

Die Klammern um das Argument werden oft weggelassen, wenn sie redundant sind.

Die Menge der Substitutionen heißt Σ. Für $\sigma, \tau \in \Sigma$ sei die Komposition $\sigma \circ \tau$ definiert durch $(\sigma \circ \tau)x := \sigma(\tau x)$.

Für $\sigma \in \Sigma$ seien Definitions- und Wertebereich (englisch: domain, co-domain) definiert:

$$\begin{array}{rcl} dom(\sigma) & := & \{\, x \,|\, x \text{ Variable } \wedge \sigma(x) \neq x \,\} \\ codom(\sigma) & := & \{\, \sigma(x) \,|\, x \text{ Variable } \wedge \sigma(x) \neq x \,\} \\ & & = \{\, \sigma(x) \,|\, x \in dom(\sigma) \,\} \end{array}$$

Σ_p *ist die Menge der* primitiven Substitutionen*:*

$$\Sigma_p := \{\, \sigma \in \Sigma \,|\, codom(\sigma) \subseteq \Lambda_p^C \,\}$$

Primitive Terme von Kontexten im allgemeinen Fall werden nun zunächst über einen Index ϱ definiert, welcher die Zuordnung für die Kontextnamen übernimmt.

Definition 4.15 (η_ϱ). *Für $\varrho \in L$ sei $\eta_\varrho : \Lambda^C \to \mathcal{P}(\Lambda)$ mit:*

$$\begin{array}{rcl} \eta_\varrho(\texttt{Bot}) & = & \Lambda_p(\texttt{Bot}) \\ \eta_\varrho(\texttt{Fun}) & = & \Lambda_p(\texttt{Fun}) \\ \eta_\varrho(\texttt{c}\ C_1 \ldots C_n) & = & \{\, t \,|\, t \Downarrow_S \texttt{c}\ t_1 \ldots t_n \wedge \forall i : t_i \in \eta_\varrho(C_i) \,\} \\ \eta_\varrho(\langle C_1, \ldots, C_n \rangle) & = & \bigcup_i \eta_\varrho(C_i) \\ \eta_\varrho(C_1 \cap \ldots \cap C_n) & = & \bigcap_i \eta_\varrho(C_i) \\ \eta_\varrho(N) & = & \varrho(N) \text{ für } N \in N(\mathcal{C}) \\ \eta_\varrho(S \texttt{ where } T = N) & = & \bigcup_{\substack{\sigma \in \Sigma_p : \Lambda_p(\sigma T) \subseteq \eta_\varrho(N) \\ \wedge dom(\sigma) \supseteq \mathcal{FV}(T)}} \eta_\varrho(\sigma S) \end{array}$$

Bemerkung. *Durch die Sonderstellung der Jecs sieht der Fall von Konstruktorkontexten besonders einfach aus, wenn ein Jec vorliegt:*

$$\eta_\varrho((C_1,\ldots,C_n)) = \{\,(t_1,\ldots,t_n)\,|\,\forall i : t_i \in \eta_\varrho(C_i)\,\}.$$

Schon an dieser Stelle können wir etwas über die Vertauschbarkeit von `where`s sagen.

Lemma 4.16 (Verschachtelte `where`s). *T und U seien Muster ohne gemeinsame Variablen. Dann gilt:*

$$\eta_\varrho((S \texttt{ where } T = N) \texttt{ where } U = M) = \eta_\varrho((S \texttt{ where } U = M) \texttt{ where } T = N)$$

Beide Kontextausdrücke sind äquivalent, und wir schreiben dafür auch S `where` $T = N; U = M$ oder S `where` $U = M; T = N$ oder S `where` $(T,U) = (N,M)$.

Beweis.

$$\begin{aligned}
\eta_\varrho((S \texttt{ where } T = N) \texttt{ where } U = M) &= \bigcup_{\substack{\sigma\in\Sigma_p:\Lambda_p(\sigma U)\subseteq\eta_\varrho(M)\\ \wedge dom(\sigma)\supseteq\mathcal{FV}(U)}} \eta_\varrho(\sigma(S \texttt{ where } T = N)) \\
&= \bigcup_{\substack{\sigma\in\Sigma_p:\Lambda_p(\sigma U)\subseteq\eta_\varrho(M)\\ \wedge dom(\sigma)\supseteq\mathcal{FV}(U)}} \left(\bigcup_{\substack{\tau\in\Sigma_p:\Lambda_p(\tau\sigma T)\subseteq\eta_\varrho(N)\\ \wedge dom(\tau)\supseteq\mathcal{FV}(\sigma T)}} \eta_\varrho(\tau\sigma S) \right) \\
&= \bigcup_{\substack{\sigma,\tau\in\Sigma_p:\Lambda_p(\sigma U)\subseteq\eta_\varrho(M)\\ \wedge\Lambda_p(\tau\sigma T)\subseteq\eta_\varrho(N)\\ \wedge dom(\sigma)\supseteq\mathcal{FV}(U)\\ \wedge dom(\tau)\supseteq\mathcal{FV}(\sigma T)}} \eta_\varrho(\tau\sigma S) \\
&= \bigcup_{\substack{\sigma,\tau\in\Sigma_p:\Lambda_p(\tau\sigma U)\subseteq\eta_\varrho(M)\\ \wedge\Lambda_p(\tau\sigma T)\subseteq\eta_\varrho(N)\\ \wedge dom(\sigma)\supseteq\mathcal{FV}(U)\\ \wedge dom(\tau)\supseteq\mathcal{FV}(\sigma T)}} \eta_\varrho(\tau\sigma S) \qquad (4.1) \\
&= \bigcup_{\substack{\pi:\exists\sigma,\tau\in\Sigma_p:\tau\circ\sigma=\pi\\ \wedge\Lambda_p(\pi U)\subseteq\eta_\varrho(M)\\ \wedge\Lambda_p(\pi T)\subseteq\eta_\varrho(N)\\ \wedge dom(\sigma)\supseteq\mathcal{FV}(U)\\ \wedge dom(\tau)\supseteq\mathcal{FV}(\sigma T)}} \eta_\varrho(\pi S) \\
&= \bigcup_{\substack{\pi\in\Sigma_p:\Lambda_p(\pi U)\subseteq\eta_\varrho(M)\\ \wedge\Lambda_p(\pi T)\subseteq\eta_\varrho(N)\\ \wedge dom(\pi)\supseteq(\mathcal{FV}(U)\cup\mathcal{FV}(T))}} \eta_\varrho(\pi S)
\end{aligned}$$

und die letzte Zeile hat eine symmetrische Form: T und N können mit U und M vertauscht werden. Man kann (N,M) in S `where` $(T,U) = (N,M)$ auch als Jec interpretieren; die natürliche Definitionserweiterung von η_ϱ auf diesen Kontext ergäbe genau die letzte Zeile.

In Schritt (4.1) wird ausgenutzt, daß $\sigma U = \tau\sigma U$ ist unter der Bedingung $dom(\sigma) \supseteq \mathcal{FV}(U)$, da σU dann keine freien Variablen besitzt.

Der letzte Schritt ist richtig, weil jedes zulässige π als eine Komposition von zwei Substitutionen $\tau := \pi_{|\mathcal{FV}(T)}$ und $\sigma := \pi_{|\mathcal{FV}(U)}$ dargestellt werden kann, so daß die Bedingungen aus der vorletzten Zeile erfüllt sind. ◇

Bemerkung. *Es werden einige Beweise in dieser Arbeit mit Induktion über Teilkontexte geführt werden. Im Fall von Kontexten der Form S `where` $T = N$ wird dabei auch oft der Bezug auf σS für alle $\sigma \in \Sigma_p$ benutzt: Diese sind zwar genaugenommen keine Unterkontexte, in einem weiteren Sinn jedoch schon und können deswegen für die Induktionsordnung benutzt werden.*

Die Abbildung ϱ kann man als eine Schätzung für die primitiven Terme von Namenskontexten ansehen. Diese Schätzung wird von Δ mittels η_ϱ verbessert:

Definition 4.17.

$$\Delta : L \to L$$
$$\Delta(\varrho) := \varrho' \text{ mit } \varrho'(N) = \eta_\varrho(C) \text{ für } N = C \in \mathcal{C}$$

Für die Existenz des Fixpunktes, den wir haben wollen, benötigen wir die Monotonie von Δ:

Lemma 4.18.

$$\varrho_1 \leq \varrho_2 \implies \Delta(\varrho_1) \leq \Delta(\varrho_2)$$

Beweis. Sei $\varrho_1 \leq \varrho_2$, also $\forall N \in N(\mathcal{C}) : \varrho_1(N) \subseteq \varrho_2(N)$. Wir zeigen durch Induktion über die Struktur von C, daß $\forall C \in \Lambda^C : \eta_{\varrho_1}(C) \subseteq \eta_{\varrho_2}(C)$.

- $C = N : \eta_{\varrho_1}(N) = \varrho_1(N) \subseteq \varrho_2(N) = \eta_{\varrho_2}(N)$
- $C \in \{\texttt{Bot}, \texttt{Fun}\} : \eta_{\varrho_1}(C) = \Lambda_p(C) = \eta_{\varrho_2}(C)$
- $C = \texttt{c}\ C_1 \ldots C_n$:

$$\begin{aligned} \eta_{\varrho_1}(C) &= \{\, t \,|\, t \Downarrow_S \texttt{c}\ t_1 \ldots t_n \wedge \forall i : t_i \in \eta_{\varrho_1}(C_i) \,\} \\ &\subseteq \{\, t \,|\, t \Downarrow_S \texttt{c}\ t_1 \ldots t_n \wedge \forall i : t_i \in \eta_{\varrho_2}(C_i) \,\} \\ &= \eta_{\varrho_2}(C) \end{aligned}$$

- $C = \langle C_1, \ldots, C_n \rangle : \eta_{\varrho_1}(C) = \bigcup_i \eta_{\varrho_1}(C_i) \subseteq \bigcup_i \eta_{\varrho_2}(C_i) = \eta_{\varrho_2}(C)$
- $C = C_1 \cap \ldots \cap C_n$: analog zur Kontextvereinigung
- $C = S\ \texttt{where}\ T = N$:
  $\eta_{\varrho_1}(C) = \bigcup_{\sigma : \Lambda_p(\sigma T) \subseteq \eta_{\varrho_1}(N)} \eta_{\varrho_1}(\sigma S) \subseteq \bigcup_{\sigma : \Lambda_p(\sigma T) \subseteq \eta_{\varrho_2}(N)} \eta_{\varrho_2}(\sigma S) = \eta_{\varrho_2}(C)$

◇

Nebenbei ergeben sich weitere Monotonien:

Korollar 4.19. *Die Funktionen $\varrho \mapsto \eta_{\Delta(\varrho)}$ und $\eta_\varrho \mapsto \eta_{\Delta(\varrho)}$ sind monoton.*

Beweis. Im Beweis zu Lemma 4.18 wird die erste Monotonie bereits gezeigt; die zweite folgt daraus wegen $\eta_{\varrho_1} \leq \eta_{\varrho_2} \implies \varrho_1 \leq \varrho_2$. Man beachte, daß die zweite Funktion tatsächlich wohldefiniert ist, da in jedem nach Definition 4.15 definiertem η_ϱ alle Informationen über ϱ enthalten sind. ◇

Definition und Lemma 4.20 (β). *Der kleinste Fixpunkt von Δ, nämlich $\beta := \mu x.\Delta(x)$, existiert.*

Beweis. Folgt aus Lemma 4.13 und Lemma 4.18 mit [DP90, Satz 4.15]. ◇

Definition 4.21 (η). *Die Menge der primitiven Terme, die von einem Kontext C repräsentiert werden, ist*

$$\eta(C) := \eta_\beta(C).$$

Sie wird auch die primitive Konkretisierung *genannt.*

Nun sind also die primitiven Terme von Kontexten definiert. Eigenschaften von η werden ab dem Abschnitt 4.5 behandelt. Eine ist jedoch so grundlegend, daß sie schon hier aufgeführt wird: nämlich, daß $\eta(C)$ tatsächlich nur aus primitiven Λ-Termen besteht, wie der Name ja sagt. Dazu wird eine Darstellung des Fixpunktes benutzt, die wir der Literatur entnehmen:

Lemma 4.22.

$$\beta = \sqcap\{\varrho | \Delta(\varrho) \leq \varrho\}.$$

$\sqcap$ ist analog zum lub die *größte untere Schranke*. Diese ist hier allerdings nicht über absteigende Ketten definiert, sondern über allgemeinen Teilmengen.

Beweis. Das Lemma folgt mit der Monotonie von Δ aus dem Fixpunktsatz von Knaster und Tarski ([DP90]). Ausgenutzt wird, daß L ein vollständiger Verband ist; die Argumentation dafür ist in der gleichen Quelle zu finden. ◇

Lemma 4.23.

$$\forall C \in \Lambda^C : \eta(C) \subseteq \Lambda_p.$$

Beweis. Wir betrachten $\varrho_p : x \mapsto \Lambda_p$. Aus Definition 4.15 ist zu erkennen, daß $\eta_{\varrho_p}(C) \subseteq \Lambda_p$ ist für jeden Kontext C, denn die Möglichkeit eines leeren Schnittes wurde dort ausgeschlossen. Dies wiederum bedeutet nach der Definition von Δ, daß auch $\Delta(\varrho_p)(C)$ immer in Λ_p enthalten ist; damit ist $\Delta(\varrho_p) \leq \varrho_p$.

Nach Lemma 4.22 bedeutet dies: $\beta \leq \varrho_p$, woraus die Aussage des Lemmas folgt. ◇

Wir sind nun bereit, die *Konkretisierung* eines Kontextes festzulegen.

4.3.2 Konkretisierung von Kontexten

Die Angabe, welche Λ-Terme denn nun ein Kontext endgültig repräsentiert, fehlt bislang. Die primitive Konkretisierung ordnet ihm nur Ausdrücke aus Λ_p zu. Für primitive Kontexte genügt das auch. Im allgemeinen sind jedoch auch unendliche Datenstrukturen erwünscht. Ist ein Kontext C mit Hilfe von Kontextnamen rekursiv definiert, so ist es möglich, daß $\eta(C)$ Datenobjekte unbeschränkter Komplexität enthält. Die Idee ist es nun, diese in aufsteigenden Ketten anzuordnen und clubs zu bilden:

Definition 4.24 (γ). *Die* Konkretisierung *eines Kontextes C ist*

$$\gamma(C) := \{\, t \mid t \equiv_c \bigsqcup_i^c s_i \text{ mit } s_1 \leq_c s_2 \leq_c \ldots \wedge \forall i : s_i \in \eta(C) \,\}.$$

Ein Beispiel wird die Bedeutung von γ illustrieren:

Lemma 4.25. *Für beliebige x ist* `repeat` $x \in \gamma(\texttt{Inf}) \setminus \eta(\texttt{Inf})$.

Beweis.

- `repeat` $x \notin \eta(\texttt{Inf})$: Es ist (`repeat` x)$\Downarrow_S$ (x`:repeat` x); der Ausdruck selbst ist also ein Teilausdruck seiner eigenen SCWHNF. Da Λ_p in Definition 3.12 als *kleinste* Menge mit den dort gegebenen Eigenschaften definiert wird, ist damit `repeat` $x \notin \Lambda_p$.

  Also ist wegen Lemma 4.23 sogar `repeat` $x \notin \eta(C)$ für beliebige Kontexte C.

- `repeat` $x \in \gamma(\texttt{Inf})$: Wie im Kapitel über Λ bereits erwähnt wurde, ist `repeat` $x \equiv_c \bigsqcup_i^c \underbrace{x\texttt{:(}\ldots x\texttt{:}}_{i\text{-mal}}\texttt{bot)}\ldots\texttt{)}$. Man kann leicht erkennen, daß alle approximierenden Terme in $\eta(\texttt{Inf})$ liegen. Damit folgt die Behauptung.

◇

Zum Schluß dieses Abschnitts erklären wir noch die Inklusion und die Konsistenz von Kontexten. Beides wird über die Konkretisierung definiert.

Definition 4.26 ($\leq$ auf Kontexten). *Seien $C, D \in \Lambda^C$.*

$$C \leq D := \gamma(C) \subseteq \gamma(D).$$

Wir sprechen von der Inklusion *oder manchmal auch von dem* Enthaltensein *von C in D .*

Definition 4.27 (Konsistenz von Kontexten). *Ein Kontext C heißt* konsistent *genau dann, wenn $\gamma(C) = \emptyset$.*

4.4 Funktionskontexte

Funktionen lassen sich mit Kontexten bis jetzt nur als `Fun` beschreiben. Diese sehr grobe, da nicht vorhandene Unterteilung steht im extremen Gegensatz zu den anderen Termen. Für jeden Λ-Term a, der keine FWHNF hat und auch keine Teilausdrücke mit FWHNF hat, gibt es nämlich nach Lemma 4.8 einen Kontext, dessen primitive Konkretisierung genau aus den nach der operationalen Semantik zu a äquivalenten Termen besteht. In diesen Fällen ist die Differenzierung der Kontexte also sehr stark. Wir wollen nun auch `Fun` weiter verfeinern, damit später die Kontextanalyse auch differenzierte Aussagen über Funktionen machen kann.

Wie man dabei vorgehen soll, wird nicht sofort klar. Schließlich lassen sich Funktionen nicht wie Konstruktoren einfach aufzählen. Sie können beliebige Namen haben und mit beliebigen Ausdrücken definiert sein. Es gibt keine einfachen Parameter, mit denen man sie konstruieren könnte.

Die grundlegende Operation, die mit Funktionen assoziiert ist, ist die Anwendung auf ein Argument. Wir können versuchen, Funktionen dadurch zu charakterisieren. Eine Funktionsanwendung kann man betrachten als (allerdings unvollständige) Spezifizierung einer Funktion durch Angabe eines Arguments und des zugehörigen Resultats.

Diese Motivation findet sich an verschiedenen Stellen:

- Im Typsystem von Damas, Hindley und Milner ([DM82]), das auch die Grundlage von Haskells Typsystem bildet, haben Funktionstypen die Form $A \to B$, wobei A der Typ des Arguments ist und B der Ergebnistyp. Diese Konstruktion könnte man auf Kontexte übertragen. Zwar sind Typen und Kontexte unterschiedliche Konzepte, doch stellen Typen ja auch in gewisser Weise Mengen von Ausdrücken dar.
- Bei einer Anfrage $f\ x \in C$ der Kontextanalyse ist die Funktion f durch die Angabe von Argument x und Ergebniskontext C charakterisiert.

Wir führen also nun, nach einer Idee von Marko Schütz, Kontexte der Form $A \overset{\forall}{\to} B$ ein. Sie sollen Funktionen aus Λ repräsentieren, die jeden Term, der von einem Kontext A beschrieben wird, abbildet auf einen der Terme, für die B steht. Analog zu Definition 4.2 sind A und B die direkten Teilkontexte von $A \overset{\forall}{\to} B$.

Diese Syntaxerweiterung entspricht in Abbildung 4.1 den zusätzlichen Produktionen

$$\begin{array}{lcl} E & \to & F \\ \overset{\forall}{\to}\text{-Kontext} \quad F & \to & E_1 \overset{\forall}{\to} E_2 \end{array}$$

Mit solchen Funktionskontexten kann viel ausgedrückt werden. Z.B. liegt eine Funktion, die einen bestimmten Wert t nach $\gamma(C)$ abbildet, in der Konkretisierung von $T \overset{\forall}{\to} C$, wobei T der primitive Kontext ist, der nur die zu t äquivalenten Ausdrücke repräsentiert (möglich, wenn T kein `Fun` enthält). Ferner kann B wiederum Terme in FWHNF repräsentieren, dann liegt eine Funktion höherer Ordnung vor, solange $\gamma(A)$ nicht leer ist. Auch A könnte selbst Funktionen repräsentieren und damit Argumente höherer Ordnung darstellen. Diese Möglichkeit werden wir jedoch aus weiter unten erklärten Gründen ausschließen müssen.

Wir definieren nun analog zu η_ϱ Funktionen η^f_ϱ, die allen Kontexten, nun inklusive der Funktionskontexte, eine Menge primitiver Terme zuweisen. Danach finden wir wieder einen Fixpunkt β^f und erklären damit ein η^f. Es wird sich herausstellen, daß sich η^f nicht von η unterscheidet auf den Termen, für die beide definiert sind (den konventionellen Kontexten).

Definition 4.28 (η^f_ϱ). *η^f_ϱ sei genauso wie η_ϱ definiert. Alle Rekursionen beziehen sich hierbei entsprechend auf η^f_ϱ anstelle von η_ϱ. Der folgende Fall kommt zur Definition neu hinzu:*

$$\eta^f_\varrho(C_1 \overset{\forall}{\to} C_2) := \{\, t \in \Lambda_p \,|\, \forall u \in \eta(C_1) : (t\ u) \in \eta^f_\varrho(C_2) \,\}$$

Bei C_1 steht in der Definition η und nicht η^f_ϱ. Warum? Auch die von den Funktionskontexten dargestellten Ausdrücke müssen wir wegen der Namen über einen Fixpunkt definieren. Lassen wir

dabei bei dem Kontext links vom „$\overset{\forall}{\to}$“ eine Rekursion zu, wird die Funktion, die den Fixpunkt besitzen soll (Δ^f), nicht mehr monoton, da eine Vergrößerung (bezüglich $\eta^f_\varrho(\cdot)$) dieses Kontextes eine mögliche Verkleinerung des Gesamtkontextes nach sich ziehen würde und umgekehrt. Um Monotonie und damit die Existenz des Fixpunktes zu gewährleisten, muß an der Stelle von C_1 also auf eine konstante Menge Bezug genommen werden: η wird benutzt.

Daraus folgt aber auch, daß links des „$\overset{\forall}{\to}$“ kein weiterer solcher Pfeil auftauchen darf, weder direkt noch durch Bezüge in Kontextnamen. Denn das gäbe wieder eine Rekursion. $\eta(C_1)$ ist nur definiert, wenn C_1 konventionell ist!

Definition 4.29. *Kontexte der Form $C_1 \overset{\forall}{\to} C_2$ sind nur zugelassen, wenn C_1 konventionell ist.*

Das stellt natürlich eine Einschränkung in der Ausdrucksfähigkeit von Kontexten dar. Wir werden dies jedoch hinnehmen, da andernfalls keine wohldefinierte Konkretisierung möglich ist — es sei denn, man kann die Existenz eines Fixpunktes bei nichtmonotonem Δ^f herleiten, was aber wesentlich schwieriger erscheint und aus dem Rahmen dieser Diplomarbeit herausfällt.

Nun definieren wir über einen neuen Fixpunkt die Menge der primitiven Terme, die von einem Kontext repräsentiert werden, der Ausdrücke der Form $A \overset{\forall}{\to} B$ enthalten und auf sie verweisen darf. Dazu benötigen wir ein neues Δ, das auch die neuen Kontexte verarbeiten kann.

Definition 4.30. *Δ^f sei analog zu Δ definiert: statt η_ϱ soll es η^f_ϱ benutzen.*

Lemma 4.31.

$$\varrho_1 \leq \varrho_2 \implies \Delta^f(\varrho_1) \leq \Delta^f(\varrho_2)$$

Beweis. Wie bei Lemma 4.18; ein Fall kommt hinzu:

- $C = C_1 \overset{\forall}{\to} C_2$:

$$\begin{aligned} \eta^f_{\varrho_1}(C) &= \{\, t \in \Lambda_p \,|\, \forall u \in \eta(C_1) : (t\ u) \in \eta^f_{\varrho_1}(C_2) \,\} \\ &\subseteq \{\, t \in \Lambda_p \,|\, \forall u \in \eta(C_1) : (t\ u) \in \eta^f_{\varrho_2}(C_2) \,\} \\ &= \eta^f_{\varrho_2}(C) \end{aligned}$$

◇

Definition 4.32. *$\beta^f := \mu x.\Delta^f(x)$ sei der kleinste Fixpunkt von Δ^f.*

Definition 4.33. *Die Menge der primitiven Terme, die von einem Kontext C repräsentiert werden, wenn Ausdrücke der Form $A \overset{\forall}{\to} B$ erlaubt sind, ist*

$$\eta^f(C) := \eta^f_{\beta^f}(C).$$

Wir werden gleich sehen, daß, wie zu erwarten ist, sich η und η^f auf den konventionellen Kontexten nicht unterscheiden. Dazu stellen wir zuerst fest, daß Lemma 4.22 direkt auf β^f übertragbar ist:

$$\beta^f = \sqcap\{\varrho | \Delta^f(\varrho) \leq \varrho\} \tag{4.2}$$

Lemma 4.34.

$$\beta = \beta^f{}_{|\Lambda^{C^0}}$$

wobei Λ^{C^0} die Menge der konventionellen Kontexte ist.

Beweis.

- „$\leq$“: Jeder Fixpunkt von Δ^f ist auch ein Fixpunkt von Δ. Denn für jedes feste ϱ können sich η_ϱ und η^f_ϱ auf konventionellen Kontexten offensichtlich nicht unterscheiden, so daß $\Delta^f(\varrho) = \varrho$ auch $\Delta(\varrho) = \varrho$ bedeutet, wenn ϱ auf konventionelle Kontexte beschränkt wird.

- „$\geq$“: Für einen Fixpunkt ϕ von Δ sei $\phi^f(N) := \phi(N)$, wenn N konventionell, und $\phi^f(N) := \eta(\texttt{Top})$ sonst. ϕ^f hat die Eigenschaft $\Delta^f(\phi^f) \leq \phi^f$, also folgt mit (4.2) $\beta^f \leq \phi^f$ und damit die Aussage.

$\diamond$

η und η^f sind also gleich auf den konventionellen Kontexten, und η^f ist nur eine Erweiterung von η. Wir werden deshalb für den Rest dieser Arbeit η und η^f miteinander **identifizieren**, ebenso auch β und β^f, Δ und Δ^f, wobei wir die Bezeichnungen ohne den Index f vorziehen. Aus dem Zusammenhang wird immer hervorgehen, ob es um allgemeine oder nur konventionelle Kontexte geht.

Zu überprüfen ist noch, ob die bereits festgestellte Eigenschaft von η, Lemma 4.23, auch für η^f gilt. Ein Blick auf die Definition 4.28 zeigt, daß dies der Fall ist; das gleiche gilt für die folgende erneute Erweiterung von η (Definition 4.35).

Die Einführung von $\overset{\forall}{\to}$-Kontexten legt nahe, analog dazu solche der Form $C_1 \overset{\exists}{\to} C_2$ zu definieren. Will man Komplemente auch von $\overset{\forall}{\to}$-Kontexten definieren (s. Abschnitt 4.8), bleibt auch gar nichts anderes übrig.

Hierbei ist es ohne weiteres möglich, sich auf der linken Seite des $\overset{\exists}{\to}$ auf η^f_ϱ zu beziehen, da die Semantik eines Existenzquantors nicht zur Zerstörung der Monotonie führt, wie es bei $\overset{\forall}{\to}$-Kontexten der Fall ist. In $C_1 \overset{\exists}{\to} C_2$ kann also auch C_1 nicht-konventionell sein.

Definition 4.35. *Analog zu Definition 4.28 sei*

$$\eta^f_\varrho(C_1 \overset{\exists}{\to} C_2) := \{\, t \in \Lambda_p \mid \exists u \in \eta^f_\varrho(C_1) : (t\ u) \in \eta^f_\varrho(C_2) \,\}$$

Die Monotonie von Δ bleibt bestehen; der Beweis in Lemma 4.18 ist für $\overset{\exists}{\to}$-Kontexte analog zum $\overset{\forall}{\to}$-Fall ergänzbar.

Mit diesen Kontexten muß man vorsichtig umgehen. Wie das folgende Beispiel zeigt, widersetzen sie sich grob der Intuition:

Die Funktionen $t_i, i \geq 0$, mit

t_{i+1} `x = case x of <1> -> []; <2> z zs -> z:`t_i `zs;`

t_0 `x = bot;`

können jeweils auf ein Argument aus $\eta(\texttt{Fin})$ angewendet werden, so daß ein Ergebnis aus $\eta(\texttt{Inf})$ resultiert. Das Argument muß nur eine genügend lange Liste sein. Sie liegen also sicher alle in $\eta(\texttt{Fin} \overset{\exists}{\to} \texttt{Inf})$ und damit auch in der Konkretisierung $\gamma(\texttt{Fin} \overset{\exists}{\to} \texttt{Inf})$, wie auch intuitiv zu erwarten ist.

Man sieht leicht, daß in der Notation von Lemma 3.32 $t_i \equiv_c f^i$ `bot` ist mit

`f r x = case x of <1> -> []; <2> z zs -> z:r;`

Ebenfalls ist anhand der Reduktionen leicht zu erkennen, daß `Y` f eine Identitätsfunktion id_{List} auf Listen darstellt. Das bedeutet aber nach Lemma 3.32, daß id_{List} der club der t_i ist, also in $\gamma(\texttt{Fin} \overset{\exists}{\to} \texttt{Inf})$ liegt, wo id_{List} intuitiv ganz sicher nicht hingehört!

Dieser drastische Fall von Verletzung der Intuition legt uns nahe, sich auf die Funktionskontexte der zuerst eingeführten Form zu konzentrieren.

Zu den primitiven Konkretisierungen $\eta(C_1 \overset{\forall}{\to} C_2)$ und $\eta(C_1 \overset{\exists}{\to} C_2)$ ist anzumerken, daß sie nicht immer nur Terme enthalten, die eine FWHNF haben. Wenn $\eta(C_1)$ leer ist, so enthält $\eta(C_1 \overset{\forall}{\to} C_2)$ nach Definition 4.28 *alle* primitiven Λ-Ausdrücke. ($\eta(C_1 \overset{\exists}{\to} C_2)$ ist in diesem Fall leer.) Wenn ferner $\eta(C_2)$ Terme aus $\Lambda_p(\texttt{Bot})$ enthält (damit ist auch immer gleich $\Lambda_p(\texttt{Bot}) \subseteq \eta(C_2)$, da $\Lambda_p(\texttt{Bot})$ nur untereinander äquivalente Terme enthält und die Definition von η nie zwischen äquivalenten Termen unterscheidet), dann enthält $\eta(C_1 \overset{\forall}{\to} C_2)$ alle Terme t, die eine SCWHNF oder keine WHNF haben: Für diese Terme gilt nämlich $(t\ u)\Uparrow$ für jedes u. Ähnliches trifft auch für $\eta(C_1 \overset{\exists}{\to} C_2)$ zu.

Wir schreiben diese Beobachtung formal für $\eta_{\Delta^i(\mathbf{0})}$ auf, da es in dieser Form im nächsten Abschnitt benutzt wird.

Lemma 4.36. *Für* $\eta_{\Delta^i(\mathbf{0})}(C_1 \overset{\forall}{\to} C_2)$ *gilt:*

$$\begin{array}{rll} \eta_{\Delta^i(\mathbf{0})}(C_1 \overset{\forall}{\to} C_2) = \Lambda_p & \textit{genau dann, wenn} & \eta(C_1) = \emptyset \\ \Lambda_p \setminus \Lambda_p(\texttt{Fun}) \subseteq \eta_{\Delta^i(\mathbf{0})}(C_1 \overset{\forall}{\to} C_2) & \textit{wenn} & \eta_{\Delta^i(\mathbf{0})}(C_2) \supseteq \Lambda_p(\texttt{Bot}) \\ \eta_{\Delta^i(\mathbf{0})}(C_1 \overset{\forall}{\to} C_2) \subseteq \Lambda_p(\texttt{Fun}) & \textit{sonst.} & \end{array}$$

Für $\eta_{\Delta^i(\mathbf{0})}(C_1 \overset{\exists}{\to} C_2)$ *gilt:*

$$\begin{array}{ll} \Lambda_p \setminus \Lambda_p(\texttt{Fun}) \subseteq \eta_{\Delta^i(\mathbf{0})}(C_1 \overset{\exists}{\to} C_2) & \textit{gdw. } \eta_{\Delta^i(\mathbf{0})}(C_1) \neq \emptyset \wedge \eta_{\Delta^i(\mathbf{0})}(C_2) \supseteq \Lambda_p(\texttt{Bot}) \\ \eta_{\Delta^i(\mathbf{0})}(C_1 \overset{\exists}{\to} C_2) \subseteq \Lambda_p(\texttt{Fun}) & \textit{sonst.} \end{array}$$

Die Aussagen treffen auch zu, wenn man überall $\eta_{\Delta^i(\mathbf{0})}$ *durch* η *ersetzt.*

Nachfolgend sind einige Implikationen im Zusammenhang mit $\overset{\forall}{\to}$-Kontexten aufgeführt. Eine Aussage der Form von (4.3) wird zwar nirgendwo in dieser Arbeit verwendet; die Formel ist aber der Vollständigkeit halber mit aufgeführt.

Lemma 4.37. *Für alle* Λ*-Ausdrücke* f *und alle Kontexte* C_1 *und* C_2*, so daß* $C_1 \overset{\forall}{\to} C_2$ *ein nach Definition 4.29 zulässiger Kontext ist, gelten die folgenden Implikationen. Ein durchgestrichener Pfeil* $\not\Rightarrow$ *bedeutet, daß die entsprechende Implikation nicht für alle* C_1, C_2, f *gilt.*

$$\forall y \in \gamma(C_1) : f\ y \in \eta(C_2) \tag{4.3}$$
$$\Downarrow \not\Uparrow$$
$$f \in \eta(C_1 \overset{\forall}{\to} C_2) \textit{ bzw. } \forall y \in \eta(C_1) : f\ y \in \eta(C_2) \tag{4.4}$$
$$\Downarrow \not\Uparrow$$
$$f \in \gamma(C_1 \overset{\forall}{\to} C_2) \tag{4.5}$$
$$\Downarrow$$
$$\forall y \in \gamma(C_1) : f\ y \in \gamma(C_2) \tag{4.6}$$
$$\Downarrow$$
$$\forall y \in \eta(C_1) : f\ y \in \gamma(C_2) \tag{4.7}$$

Beweis.

- (4.4) $\not\Longrightarrow$ (4.3): Z.B. für $C_1 = C_2 = \texttt{Inf}$ und $f = \texttt{id}$.
- (4.5) $\not\Longrightarrow$ (4.4): Z.B. für $C_1 = C_2 = \texttt{Inf}$ und $f = \texttt{const (repeat 1)}$. Mit $f_i = \texttt{const ((1:)}^i\texttt{ bot)} \in \eta(C_1 \overset{\forall}{\to} C_2)$ ist $f \equiv_c \bigsqcup_i^c f_i$.
- (4.5) $\Longrightarrow$ (4.6): Sei $f \in \gamma(C_1 \overset{\forall}{\to} C_2)$. Es ist $f \equiv_c \bigsqcup_i^c f_i$ mit $\forall i : \forall x \in \eta(C_1) : f_i\ x \in \eta(C_2)$. Sei nun $y \in \gamma(C_1), y \equiv_c \bigsqcup_i^c y_i$ mit $y_i \in \eta(C_1)$. Dann ist $f\ y \equiv_c (\bigsqcup_i^c f_i)\ (\bigsqcup_j^c y_j) \equiv_c \bigsqcup_i^c \underbrace{(f_i\ y_i)}_{\in \eta(C_2)} \in \gamma(C_2)$ (nach Lemma 3.33).

Die restlichen Implikationen folgen sofort aus der Definition von γ. ◇

Bemerkung.

1. *Wenn* $\gamma(C_2)$ *club-abgeschlossen ist, gilt sogar* (4.7) $\Longrightarrow$ (4.6).
2. *Eine häufige Anwendung des Lemmas ist die Folgerung* (4.4) $\Longrightarrow$ (4.6) *ohne die Zwischenstufe* (4.5) *und ohne explizite Angabe einer Funktion* f*. Statt dessen ist ein Term angegeben, der von* y *abhängt und hier* $f\ y$ *entspricht. Würde man* f *explizit aufschreiben, käme dadurch oft eine „verbotene" Funktion zum Vorschein, die Jec-Konstrukte benutzt. Das verursacht aber keine weiteren Probleme, da auch solche „verbotenen" Funktionen stetig sind.*
3. *Ob* (4.6) $\Longrightarrow$ (4.5) *gilt, ist nicht klar. Gegenbeispiele sind bis jetzt noch nicht gefunden worden. Diese Implikation ist für die Korrektheit der Analyse in Verfahren 6.2 von Bedeutung.*

4.5 Konstruktion von η

Wir wollen in diesem Abschnitt zeigen, daß die primitiven Terme eines konventionellen Kontextes C iterativ erzeugt werden können und $\eta(C)$ durch $\bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)$ gegeben ist.

Das folgende Lemma wird uns nachher helfen, eine Schwierigkeit bei `where`-Kontexten zu überwinden. Es sagt aus, daß die Werte eines primitiven Kontextes, die in $\bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)$ für ein beliebiges C enthalten sind, schon in einer der Approximationen $\eta_{\Delta^i(\mathbf{0})}(C)$ liegen müssen.

Lemma 4.38. *Sei T ein primitiver Kontext und C ein beliebiger Kontext.*

$$\Lambda_p(T) \subseteq \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C) \implies \exists i_0 : \Lambda_p(T) \subseteq \eta_{\Delta^{i_0}(\mathbf{0})}(C)$$

Beweis. Zur Behandlung der nicht-konventionellen Kontexte werden zuerst einige Vorbereitungen getroffen:

Sei $y \in \Lambda \setminus \Lambda_p$ beliebig, aber fest. C habe für den Augenblick die Form $C_1 \overset{\forall}{\to} C_2$ oder $C_1 \overset{\exists}{\to} C_2$. Weiter sei angenommen, daß nicht der Fall aus Lemma 4.36 eintritt, bei dem $\eta_{\Delta^i(\mathbf{0})}(C) = \Lambda_p$ ist. Dann ist mit `const` y eine Funktion gegeben, die nicht in $\eta_{\Delta^i(\mathbf{0})}(C)$ enthalten ist; der Grund ist, daß $\eta_{\Delta^i(\mathbf{0})}(C)$ über $\eta_{\Delta^i(\mathbf{0})}(C_2)$ definiert ist, worin y sicher nicht liegen kann.

Wir setzen $F := \Lambda_p(\texttt{Fun}) \setminus \{\texttt{const}\ y\}$.

F^* sei die kleinste Menge, für die gilt:

$$F^* = F \cup \{\, x \,|\, x \Downarrow_S \texttt{c}\ x_1 \ldots x_n \wedge \texttt{c} \text{ ein Konstruktor} \wedge \forall j : x_j \in F^* \,\}$$

Sei nun $M_{T,i,C} := \Lambda_p(T) \cap \eta_{\Delta^i(\mathbf{0})}(C) \setminus F^*$. Wir zeigen, daß für alle $T \in \Lambda_p^C$, $i \geq 0$ und $C \in \Lambda^C$ gilt:

$$M_{T,i,C} \neq \emptyset \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C) \tag{4.8}$$

Die Aussage des Lemmas folgt dann hieraus, da für jedes T und C mit $\Lambda_p(T) \subseteq \bigcup_j \eta_{\Delta^j(\mathbf{0})}(C)$ ein i gibt, so daß $M_{T,i,C} \neq \emptyset$. Denn $\Lambda_p(T) \setminus F^*$ ist nie leer: Das käme offenbar höchstens für ein T in Frage, das nur aus `Fun` und Konstruktoren besteht. In diesem Fall aber ist $t \in \Lambda_p(T) \setminus F^*$, wobei der Λ-Term t aus T entsteht, indem jedes Vorkommen von `Fun` durch `const` y ersetzt wird.

Der Beweis von (4.8) läuft per Induktion über die Struktur von C und über i. Die Ordnung, mit der die Induktion funktioniert, ist diejenige mit $(i_0, C) < (i_0', C') :\iff i_0 < i_0' \vee (i_0 = i_0' \wedge C$ ist echter Teilkontext von $C')$.

- $C = N$:
  - $i = 0$: $\eta_{\Delta^0(\mathbf{0})}(N) = \emptyset$, also ist $M_{T,i,C}$ leer und nichts ist zu zeigen.
  - $i > 0$: Sei $N = D \in \mathcal{C}$.
    $\eta_{\Delta^i(\mathbf{0})}(N) = \eta_{\Delta^{i-1}(\mathbf{0})}(D)$; somit ist $M_{T,i,N} = M_{T,i-1,D}$ und $M_{T,i,N} \neq \emptyset \implies \Lambda_p(T) \subseteq \eta_{\Delta^{i-1}(\mathbf{0})}(D) = \eta_{\Delta^i(\mathbf{0})}(N)$.
- $C =$ `Bot`: T kann nur `Bot` sein, also $\Lambda_p(T) = \eta_{\Delta^i(\mathbf{0})}(C)$.
- $C =$ `Fun`: T kann nur `Fun` sein, also $\Lambda_p(T) = \eta_{\Delta^i(\mathbf{0})}(C)$.
- $C = \langle C_1, \ldots, C_n \rangle$: klar für $n = 0$; andernfalls ist $M_{T,i,C} \neq \emptyset \implies \exists j : M_{T,i,C_j} \neq \emptyset$ und $M_{T,i,C_j} \neq \emptyset \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C_j) \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C)$.
- $C = C_1 \cap \ldots \cap C_n$: Es ist $\forall j : M_{T,i,C} \subseteq M_{T,i,C_j}$ und $M_{T,i,C_j} \neq \emptyset \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C_j)$, somit $M_{T,i,C} \neq \emptyset \implies \forall j : M_{T,i,C_j} \neq \emptyset \implies \forall j : \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C_j) \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C)$.
- $C = S$ `where` $U = N$: $M_{T,i,C} \neq \emptyset \implies \exists \sigma : M_{T,i,\sigma S} \neq \emptyset$ und $M_{T,i,\sigma S} \neq \emptyset \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(\sigma S) \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C)$.

- $C = \mathtt{c}\ C_1 \ldots C_n$: T kann nur die Form $\mathtt{c}\ T_1 \ldots T_n$ haben. So, wie F^* definiert ist, überträgt sich auch die Struktur von F^* auf Argumente eines Konstruktors. Damit ist $M_{T,i,C} = \{x | x \Downarrow_S \mathtt{c}\ x_1 \ldots x_n \wedge \forall j : x_j \in M_{T_j,i,C_j}\}$. Es folgt $M_{T,i,C} \neq \emptyset \implies \forall j : M_{T_j,i,C_j} \neq \emptyset \implies \forall j : \Lambda_p(T_j) \subseteq \eta_{\Delta^i(\mathbf{0})}(C_j) \implies \Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C)$.

- $C = C_1 \overset{\forall}{\to} C_2$: In diesem und in dem nächsten Fall wird Gebrauch von Lemma 4.36 gemacht.

  Ist $\eta(C_1) = \emptyset$ oder $\eta_{\Delta^i(\mathbf{0})}(C_2) \supseteq \Lambda_p(\mathtt{Bot})$, so ist für $T \neq \mathtt{Fun}$: $\Lambda_p(T) \subseteq \Lambda_p \setminus \Lambda_p(\mathtt{Fun}) \subseteq \eta_{\Delta^i(\mathbf{0})}(C)$.

  Ist $T = \mathtt{Fun}$ und $\eta(C_1) = \emptyset$, so ist $\Lambda_p(T) \subseteq \eta_{\Delta^i(\mathbf{0})}(C) = \Lambda_p$.

  Wenn $T = \mathtt{Fun}$, $\eta(C_1) \neq \emptyset$ und $\eta_{\Delta^i(\mathbf{0})}(C_2) \supseteq \Lambda_p(\mathtt{Bot})$, so ist $\Lambda_p(T) \cap \eta_{\Delta^i(\mathbf{0})}(C) \subseteq F$, also $M = \emptyset$ und nichts zu zeigen.

  In sonstigen Fällen ist immer $\eta_{\Delta^i(\mathbf{0})}(C) \setminus F = \emptyset$ und damit $M = \emptyset$ und nichts zu zeigen.

- $C = C_1 \overset{\exists}{\to} C_2$: Falls $\eta_{\Delta^i(\mathbf{0})}(C_2) \supseteq \Lambda_p(\mathtt{Bot})$ und $\eta_{\Delta^i(\mathbf{0})}(C_1) \neq \emptyset$ sowie $T \neq \mathtt{Fun}$, ist $\Lambda_p(T) \subseteq \Lambda_p \setminus \Lambda_p(\mathtt{Fun}) \subseteq \eta_{\Delta^i(\mathbf{0})}(C)$.

  Bei $T = \mathtt{Fun}$ und sonst gleichen Bedingungen ist $\Lambda_p(T) \cap \eta_{\Delta^i(\mathbf{0})}(C) \subseteq F$, also $M = \emptyset$ und nichts zu zeigen.

  Ansonsten ist $\eta_{\Delta^i(\mathbf{0})}(C) \setminus F = \emptyset$ und damit $M = \emptyset$ und nichts zu zeigen.

◇

Lemma 4.39. *$\bigsqcup_i \Delta^i(\mathbf{0})$ ist ein Fixpunkt von Δ, wenn $\overset{\forall}{\to}$-Kontexte nicht bzw. nur eingeschränkt verwendet werden.*

Beweis. Sei $F = \bigsqcup_i \Delta^i(\mathbf{0})$. Zu zeigen ist $\forall N \in N(\mathcal{C}) : F(N) = (\Delta(F))(N)$. Die „$\subseteq$"-Richtung ist trivial wegen der Monotonie von Δ. Wegen
$F(N) = \bigcup_i (\Delta^i(\mathbf{0})(N)) = \mathbf{0}(N) \cup \bigcup_i (\Delta^{i+1}(\mathbf{0})(N)) = \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)$ und
$(\Delta(F))(N) = \eta_F(C)$ (für $N = C \in \mathcal{C}$) bleibt zu zeigen:

$$\forall C \in \Lambda^C : \eta_F(C) \overset{!}{\subseteq} \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)$$

Wir führen eine Induktion über die Struktur von C durch:

- $C = N$: $\eta_F(N) = (\bigsqcup_i \Delta^i(\mathbf{0}))(N) = \bigcup_i (\Delta^i(\mathbf{0})(N)) = \bigcup_i \eta_{\Delta^i(\mathbf{0})}(N)$
- $C \in \{\mathtt{Bot}, \mathtt{Fun}\}$: $\eta_F(C) = \Lambda_p(C) = \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)$
- $C = \mathtt{c}\ C_1 \ldots C_n$:

$$\begin{aligned}
\eta_F(C) &= \{\, t \mid t \Downarrow_S \mathtt{c}\ t_1 \ldots t_n \wedge \forall j : t_j \in \eta_F(C_j) \,\} \\
&= \{\, t \mid t \Downarrow_S \mathtt{c}\ t_1 \ldots t_n \wedge \forall j : t_j \in \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C_j) \,\} \\
&\subseteq \bigcup_i \{\, t \mid t \Downarrow_S \mathtt{c}\ t_1 \ldots t_n \wedge \forall j : t_j \in \eta_{\Delta^i(\mathbf{0})}(C_j) \,\} \qquad (4.9) \\
&= \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)
\end{aligned}$$

Schritt (4.9) sieht man folgendermaßen[1]:
$\forall j : t_j \in \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C_j) \implies \forall j : \exists i_j : t_j \in \eta_{\Delta^{i_j}(\mathbf{0})}(C_j) \implies \exists i_{\max} : \forall j : t_j \in \eta_{\Delta^{i_{\max}}(\mathbf{0})}(C_j)$.

[1] Hier wird die Monotonie von Δ ausgenutzt: $i_1 \leq i_2 \implies \Delta^{i_1}(\mathbf{0}) \leq \Delta^{i_2}(\mathbf{0}) \implies \eta_{\Delta^{i_1}(\mathbf{0})}(C_j) \subseteq \eta_{\Delta^{i_2}(\mathbf{0})}(C_j)$

- $C = \langle C_1, \ldots, C_n \rangle$: $\eta_F(C) = \bigcup_j \eta_F(C_j) = \bigcup_j \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C_j) = \bigcup_i \bigcup_j \eta_{\Delta^i(\mathbf{0})}(C_j) = \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)$
- $C = C_1 \cap \ldots \cap C_n$: Analog zur Kontextvereinigung; der Schritt $\bigcap_j \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C_j) \subseteq \bigcup_i \bigcap_j \eta_{\Delta^i(\mathbf{0})}(C_j)$ ist wie im Fall der Konstruktorkontexte nachzuvollziehen.
- $C = S$ `where` $T = N$:

$$\begin{aligned}
\eta_F(C) &= \bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \eta_F(N)} \eta_F(\sigma S) \\
&= \bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \bigcup_{i_1} \eta_{\Delta^{i_1}(\mathbf{0})}(N)} \bigcup_{i_2} \eta_{\Delta^{i_2}(\mathbf{0})}(\sigma S) \\
&= \bigcup_{i_2} \bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \bigcup_{i_1} \eta_{\Delta^{i_1}(\mathbf{0})}(N)} \eta_{\Delta^{i_2}(\mathbf{0})}(\sigma S) \\
&\subseteq \bigcup_{i_2} \bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \eta_{\Delta^{i_2}(\mathbf{0})}(N)} \eta_{\Delta^{i_2}(\mathbf{0})}(\sigma S) \qquad (4.10) \\
&= \bigcup_{i_2} \eta_{\Delta^{i_2}(\mathbf{0})}(C)
\end{aligned}$$

  Hierbei ist Schritt (4.10) eine Anwendung von Lemma 4.38.

- $C = C_1 \xrightarrow{\exists} C_2$: $\begin{aligned}[t]
\eta_F(C) &= \{\, t \in \Lambda_p \mid \exists u \in \eta(C_1) : (t\ u) \in \eta_F(C_2) \,\} \\
&= \{\, t \in \Lambda_p \mid \exists u \in \eta(C_1) : (t\ u) \in \textstyle\bigcup_i \eta_{\Delta^i(\mathbf{0})}(C_2) \,\} \\
&= \textstyle\bigcup_i \{\, t \in \Lambda_p \mid \exists u \in \eta(C_1) : (t\ u) \in \eta_{\Delta^i(\mathbf{0})}(C_2) \,\} \\
&= \textstyle\bigcup_i \eta_{\Delta^i(\mathbf{0})}(C)
\end{aligned}$

- $C = C_1 \xrightarrow{\forall} C_2$: Der entscheidende zu zeigende Schritt ist

$$\begin{aligned}
& \{\, t \in \Lambda_p \mid \forall u \in \eta(C_1) : (t\ u) \in \bigcup_i \eta_{\Delta^i(\mathbf{0})}(C_2) \,\} \\
\subseteq\ & \bigcup_i \{\, t \in \Lambda_p \mid \forall u \in \eta(C_1) : (t\ u) \in \eta_{\Delta^i(\mathbf{0})}(C_2) \,\}
\end{aligned}$$

  Das gilt sicher für den Fall, daß $\eta(C_1)$ endlich ist — was aber gleichbedeutend mit $\eta(C_1) = \emptyset$ ist, denn mit einem Term t enthält $\eta(C_1)$ immer auch alle zu t äquivalenten Terme, da die Definition von η und η_ϱ nie zwischen diesen unterscheiden, und das sind unendlich viele. Andererseits gilt die Ungleichung auch, wenn C_1 nur aus `Bot` und Konstruktoren besteht: Nach Lemma 4.8 ist dann die Anzahl der Äquivalenzklassen bezüglich $\equiv_c$, welche von $\eta(C_1)$ umfaßt werden, endlich. In diesem Fall können die $(t\ u)$ durch Lemma 3.24 also nur endlich viele Werte modulo $\equiv_c$ annehmen, und dann beweisen jenes Lemma 4.8 und der weiter unten aufgeführte Satz 4.41, daß die Behauptung gilt. Zwar benutzt letzteres Lemma die Ergebnisse aus diesem Abschnitt und damit auch die Aussage, die wir gerade beweisen — ein Trugschluß, ein „Beweis durch sich selbst“? Nein, denn wir gebrauchen das Lemma 4.41 nur für C_1 aus `Bot` und Konstruktoren, und diesen Fall berührt der aktuelle Unterpunkt nicht.

  Die Ungleichung gilt aber sicher nicht immer: Wenn $C_1 = C_2 =$ `Fin` ist, liegt die Identitätsfunktion `id` nach diesem Lemma offenbar in der ersten Menge. Es gibt aber kein i, so daß $\eta($`Fin`$) = \eta_{\Delta^i(\mathbf{0})}($`Fin`$)$ wäre, da dieses j unter anderem ein Maß für die Länge der längsten Listen ist, die in $\eta_{\Delta^i(\mathbf{0})}($`Fin`$)$ enthalten sind; somit ist `id` nicht in der zweiten Menge enthalten.

◇

Bei dem so gefundenen Fixpunkt handelt es sich nun tatsächlich um den kleinsten, also für η relevanten Fixpunkt:

Korollar 4.40.

$$\beta_{|\Lambda^{C^0}} = \bigsqcup_i \Delta^i(\mathbf{0}),$$

wobei Λ^{C^0} die Menge der konventionellen Kontexte sei.

Beweis. Wie eben gesehen, ist $\bigsqcup_i \Delta^i(\mathbf{0})$ ein Fixpunkt. Es ist auch der kleinste, denn für ein beliebiges F mit $\Delta(F) = F$ folgt aus $\mathbf{0} \leq F$ und der Monotonie von Δ induktiv $\bigsqcup_i \Delta^i(\mathbf{0}) \leq F$. $\diamond$

Die nun gefundene Eigenschaft der konventionellen Kontexte bedeutet auch, daß nichtkonventionelle Kontexte eine durchaus kompliziertere Semantik haben als diese. Manche Sätze in den Analyseverfahren werden daher die Einschränkung benötigen, daß gewisse Kontexte konventionell sind.

4.6 Weitere Sätze über η und γ

Die Sätze in diesem Abschnitt werden uns später behilflich sein, wenn es um die Analyseverfahren geht.

Das nächste Lemma sagt aus, daß — auf die primitive Konkretisierung bezogen — jeder konventionelle Kontext einen beliebigen primitiven Kontext entweder ganz umfaßt oder disjunkt zu ihm ist. Für konventionelle C ist somit $\eta(C)$ eine disjunkte Vereinigung von $\eta(P)$ mit $P \in \Lambda_p^C$.

Lemma 4.41. *Sei C ein konventioneller Kontext und T ein primitiver Kontext. Dann gilt*

$$(\exists x \in \Lambda_p(T) : x \in \eta(C)) \implies \Lambda_p(T) \subseteq \eta(C)$$

Beweis. Es genügt zu zeigen $\forall i : [(\exists x \in \Lambda_p(T) : x \in \eta_{\Delta^i(\mathbf{0})}(C)) \implies \Lambda_p(T) \subseteq \eta(C)]$. Dies geschieht durch Induktion über i und die Struktur von C mit der Induktionsordnung $(i, C) < (i', C') \iff (i < i' \vee (i = i' \wedge C$ ist echter Teilkontext von $C'))$.

- $C = N$:
  - $i = 0$: $x \in \eta_{\Delta^0(\mathbf{0})}(C) = \emptyset$ ist unmöglich und nichts ist zu zeigen.
  - $i > 0$: Sei $N = D \in \mathcal{C}$. Es gilt $x \in \eta_{\Delta^i(\mathbf{0})}(N) = \eta_{\Delta^{i-1}(\mathbf{0})}(D) \implies \Lambda_p(T) \subseteq \eta(D) = \eta(N)$.
- $C \in \{\texttt{Bot}, \texttt{Fun}\}$: Wenn $x \in \eta_{\Delta^i(\mathbf{0})}(C)$, kann T nur `Bot` bzw. `Fun` sein, und $\Lambda_p(T) \subseteq \eta(C)$ gilt.
- $C = \langle C_1, \ldots, C_n \rangle$ oder $C = C_1 \cap \ldots \cap C_n$: Die Induktion folgt sofort über einen bzw. alle Teilkontexte $C_1, \ldots, C_n$.
- $C = \texttt{c}\ C_1 \ldots C_n$: Aus $x \in \eta_{\Delta^i(\mathbf{0})}(C)$ folgt: T hat die Form $\texttt{c}\ T_1 \ldots T_n$, $x \Downarrow_S \texttt{c}\ x_1 \ldots x_n$, $\forall j : x_j \in \Lambda_p(T_j) \cap \eta_{\Delta^i(\mathbf{0})}(C_j)$, $\forall j : \Lambda_p(T_j) \subseteq \eta(C_j)$ und schließlich $\Lambda_p(T) \subseteq \eta(C)$.
- $C = S$ `where` $P = N$: Die Induktion folgt sofort über σS für ein σ.

$\diamond$

Korollar 4.42. *Sei C ein konventioneller Kontext.*

$$\eta(x \texttt{ where } x = C) = \eta(C).$$

Beweis. Nach der Definition von η im `where`-Fall gilt „$\subseteq$“. Für die andere Richtung beachten wir, daß es für jedes $z \in \Lambda_p$ einen primitiven Kontext P_z gibt mit $z \in \Lambda_p(P_z)$. Also läßt sich für $z \in \eta(C)$ Lemma 4.41 anwenden mit dem Ergebnis $\Lambda_p(P_z) \subseteq \eta(C)$; daraus folgt $z \in \eta(x \texttt{ where } x = C)$. $\diamond$

Das folgende Lemma sagt aus, daß alle primitiven Λ-Ausdrücke in der Konkretisierung eines konventionellen Kontextes C bereits in $\eta(C)$ liegen müssen.

Lemma 4.43. *Sei C ein konventioneller Kontext. Es gilt*

$$\gamma(C) \cap \Lambda_p = \eta(C).$$

Beweis.
- $\supseteq$: Klar mit der Definition von γ und Lemma 4.23.
- $\subseteq$: Für $a \in \Lambda_p$ gibt es einen primitiven Kontext A mit $a \in \Lambda_p(A)$ (Lemma 4.7). Wenn außerdem $a \in \gamma(C)$, ist $a \equiv_c \bigsqcup_i^c b_i$ mit $\forall i : b_i \in \eta(C)$. Nach Lemma 4.9 gibt es dann ein $b_i \in \Lambda_p(A)$. Schließlich läßt sich Lemma 4.41 anwenden; $a \in \eta(C)$ folgt.

◇

Nachdem nun schon einige Hilfsmittel aufgestellt sind, können wir jetzt recht schnell feststellen, daß Teilmengenbeziehungen bei konventionellen Kontexten unabhängig davon sind, ob sie mittels η oder γ formuliert sind.

Lemma 4.44. *Sei C_1 ein beliebiger und C_2 ein konventioneller Kontext. Dann gilt*

$$\gamma(C_1) \subseteq \gamma(C_2) \iff \eta(C_1) \subseteq \eta(C_2)$$

Beweis.
- „$\Longleftarrow$“: Klar wegen der Definition von γ.
- „$\Longrightarrow$“: Aus $\eta(C_1) \subseteq \gamma(C_1) \subseteq \gamma(C_2)$ und $\eta(C_1) \subseteq \Lambda_p$ folgt $\eta(C_1) \subseteq \gamma(C_2) \cap \Lambda_p$. Mit Lemma 4.43 ergibt sich $\eta(C_1) \subseteq \eta(C_2)$.

◇

Schließlich zeigen wir ein Analogon zu Lemma 4.9: Die Approximationen eines clubs, der in $\eta(C)$ für ein konventionelles C enthalten ist, liegen außer eventuell einigen Anfangsgliedern alle selbst in $\eta(C)$.

Lemma 4.45. *Sei C ein konventioneller Kontext.*

$$a \in \eta(C) \wedge a \equiv_c \bigsqcup_i^c a_i \implies \exists i_0 : \forall i > i_0 : a_i \in \eta(C)$$

Beweis. $a \in \eta(C)$ impliziert $a \in \Lambda_p$, also $a \in \Lambda_p(P)$ für $P \in \Lambda_p^C$. Lemma 4.9 bezeugt nun für alle $i > i_0$, daß $a_i \in \Lambda_p(P)$. Lemma 4.41 besagt, daß dann auch $a_i \in \eta(C)$. ◇

4.7 Projektionen

Die Analyse einer Anfrage $f(x_1, \ldots, x_n) \in C_1 \overset{\forall}{\to} C_2$ oder $f(x_1, \ldots, x_n) \in C_1 \overset{\exists}{\to} C_2$ [2] hängt stark mit der Bildung von *Projektionen* zusammen, d.h. der Abbildung eines Kontextes auf einige seiner Bestandteile. Das kann man nachvollziehen, wenn man bedenkt, daß — abgesehen vom Sonderfall $\gamma(C_1) = \emptyset$ — jede Kombination $(t_1, \ldots, t_n)$ von Λ-Termen, die $f(t_1, \ldots, t_n) \in \gamma(C_1 \overset{\forall}{\to} C_2)$ oder $f(t_1, \ldots, t_n) \in \gamma(C_1 \overset{\exists}{\to} C_2)$ erfüllt, ein Teil eines Tupels $(t_1, \ldots, t_{n+1})$ ist, für das $f(t_1, \ldots, t_n)\ t_{n+1} \in \gamma(C_2)$ gilt. Diese Art von Projektion operiert auf Jecs, daneben finden auch Projektionen für andere Konstruktoren Anwendung, nämlich in einem Fall der $C_1 \overset{\forall}{\to} C_2$-Analyse, in dem es um die Inklusion von Konstruktorkontexten geht.

Aus diesem Grund führen wir in diesem Abschnitt einige Konzepte und Notationen für Projektionen ein.

[2] Für die Notation $f(\cdot, \ldots, \cdot)$ siehe Abschnitt 5.1 auf Seite 49.

4.7.1 Überblick

Es werden zwei Arten von Projektionen eingeführt:

- In Abschnitt 4.7.3 geht es um syntaktische Bestandteile eines Kontextes. So ist z.B. (`Bot`, `Inf`) das Resultat einer syntaktischen Projektion von (`Bot`, `[]`, `Inf`) auf die erste und dritte Jec-Komponente.
- In Abschnitt 4.7.4 werden Projektionen definiert, die Ähnliches tun, aber mehr auf die Semantik der Kontexte achten. Wird mit einer Projektion jenes Abschnittes z.B. auf die erste Jec-Komponente von (`Bot`, $\langle\rangle$) projiziert, so ergibt das $\langle\rangle$, da der Ausgangsterm eine leere Konkretisierung hat. Dabei wird das Ziel verfolgt, daß die primitiven Terme einer solchen Projektion aus den gewünschten Bestandteilen der primitiven Terme des Ursprungskontextes bestehen.

  Die zwei Kontextprojektionen jenes Abschnittes heißen *großzügig* bzw. *kleinlich*, weil sie sich nur in der Strenge ihrer Definition, d.h. in der Größe ihres Definitionsbereiches unterscheiden.

  In Abschnitt 4.7.5 werden die Projektionen dieser Art auf `where`-Kontexten fortgesetzt.

Da wir im Kapitel über die Analyseverfahren sehr oft über Teile von Jecs sprechen müssen, werden dort Projektionen beider Arten hauptsächlich bezüglich Jecs benutzt werden. Die Symbole Π und π werden hier eigens dafür eingeführt. Es gibt aber einen Fall innerhalb einer rekursiven $\overset{\forall}{\rightarrow}$-Analyse, bei dem die (kleinliche) Projektion in bezug auf einen beliebigen Konstruktor konstruiert werden muß.

Auch über Teile von Λ-Termen muß in Kapitel 6 oft gesprochen werden. Deshalb werden in den Abschnitten 4.7.3 und 4.7.4 zusätzlich die syntaktische Projektion für Λ-Ausdrücke definiert und die großzügige Projektion für Λ-Teilmengen.

4.7.2 Grundlegende Definitionen

Wir beginnen mit einigen Grundlagen. Bei Projektionen geht es um Bestandteile von Kontexten und der dazugehörigen Λ-Ausdrücke; man muß also in die *Struktur* von Termen hineingreifen. Eine solche Struktur wird meistens in der Gestalt von Kontextmustern daherkommen, kann aber ein beliebiger primitiver Kontext mit Variablen sein. Die Variablen stehen hier für „don't care"-Symbole. Die folgende Definition führt einen einfachen Strukturbegriff formal ein:

Definition 4.46 (S-strukturierte Kontexte und Λ-Terme). *Sei S ein Kontext, der aus* `Bot`, `Fun`, *Konstruktoren und Variablen besteht.*
Ein Kontext C ist S-strukturiert, *wenn er die Form $C \equiv \sigma S$ für $\sigma \in \Sigma$ hat.*
Ein Λ-Ausdruck a ist

`Bot`-strukturiert,	*wenn* $a \in \Lambda_p$(`Bot`).
`Fun`-strukturiert,	*wenn* $a \in \Lambda_p$(`Fun`).
`c` $S_1 \ldots S_n$-strukturiert,	*wenn a die Form* `c` $a_1 \ldots a_n$ *hat und jedes a_i S_i-strukturiert ist.*
x-strukturiert, x *Variable,*	*immer.*

Variablen in S stehen für Stellen, die von einer Projektion entweder *zurückgegeben* oder *ignoriert* werden. Um zwischen diesen beiden Möglichkeiten zu unterscheiden, benutzen wir folgende

Definition 4.47 (V^1, V^*). *Die Menge aller Kontextvariablen sei in zwei disjunkte Mengen unterteilt: Variablen aus V^1 stehen für Teilausdrücke, die eine Projektion zurückgeben soll. Variablen aus V^* stehen für von einer Projektion zu ignorierende Stellen.*

und notieren dies gegebenenfalls wie folgt:

Schreibweise. Mit einer 1 indizierte Variablen (x^1) seien aus V^1; gesternte Variablen (x^*) kommen aus V^*.

Gebraucht wird außerdem noch eine Methode, deutlich zu machen, daß im Zusammenhang mit den Projektionen eine Variable für jeden beliebigen Wert stehen soll:

Definition 4.48 ($S_{\mathtt{Top}}$). *Sei $S \in \Lambda^{C\#}$.*

$$S_{\mathtt{Top}} := \tau S \text{ mit } \tau \in \Sigma,\ \tau : x \mapsto \mathtt{Top}.$$

4.7.3 Syntaktische Projektionen auf S-strukturierten Kontexten

Die in diesem Unterabschnitt definierten Projektionen sind sehr einfach: sie liefern einen Teilkontext ihres Arguments, oder mehrere davon in einem Jec zusammengefaßt, betreffen also nur die Syntax von Kontexten. Parameter ist ein Kontextmuster S mit paarweise verschiedenen Variablen; diese sind nach der oben erklärten Notation gekennzeichnet.

Für Λ-Terme erklären wir danach eine entsprechende Projektion, die allerdings nicht rein syntaktisch operiert.

Definition 4.49 (Syntaktische Kontextprojektionen: P^s_S, Π_S). *Die* syntaktische Kontextprojektion P^s_S, *S ein Kontextmuster, ist nur auf S-strukturierten Kontexten C definiert, und zwar wie folgt:*

$$\begin{aligned}
P^s_{x^1}(C) &:= C \\
P^s_{x^*}(C) &\text{ ist nicht definiert} \\
P^s_{\mathtt{Bot}}(C) &\text{ ist nicht definiert} \\
P^s_{\mathtt{C}\ S_1 \ldots S_n}(\mathtt{c}\ C_1 \ldots C_n) &:= (P^s_{S_{i_1}}(C_{i_1}), \ldots, P^s_{S_{i_m}}(C_{i_m})) \\
&\quad \text{mit } \{S_{i_1}, \ldots, S_{i_m}\} = \{S_i | S_i \text{ enthält mind. eine Variable aus } V^1\} \\
&\quad \text{und } i_1 < \cdots < i_m.
\end{aligned}$$

Da nur die Teile zurückgegeben werden sollen, die den V^1-Variablen in S entsprechen, muß die Projektion für $S \equiv x^*$ oder $S \equiv \mathtt{Bot}$ nicht definiert sein.

Π_S wird die Projektion genannt, wenn S ein Jec ist.

Für spezielle S bekommt die Projektion einen speziellen Namen, nämlich wenn bei Konstruktorkontexten auf eine einzelne Stelle abgebildet wird oder auf alle Stellen außer einer abgebildet wird, also eine Konstruktorkomponente wegfällt:

$$\begin{aligned}
P^s_{i,\mathtt{C}} &:= P^s_S \text{ mit } S = \mathtt{c}\ \underbrace{x^*_1 \ldots x^*_n}_{x^1_i \text{ statt } x^*_i} \\
P^s_{\hat{i},\mathtt{C}} &:= P^s_S \text{ mit } S = \mathtt{c}\ \underbrace{x^1_1 \ldots x^1_n}_{x^*_i \text{ statt } x^1_i}.
\end{aligned}$$

Analog dazu verstehen sich Π_i und $\Pi_{\hat{i}}$.

Zum Beispiel ist $\Pi_2((\mathtt{Fin}, \mathtt{Bot})) = \Pi_{(x^*, y^1)}((\mathtt{Fin}, \mathtt{Bot})) = \mathtt{Bot}$.

Die Projektionen seien auch auf Λ-Ausdrücken definiert. Hier ist deren Definition allerdings nicht mehr rein syntaktischer Natur, wenn Bot in S als Unterterm vorkommt, da S-strukturierte Λ-Terme dann auch mit Hilfe von $\Lambda_p(\mathtt{Bot})$ definiert sind.

Definition 4.50 (P^s_S auf Λ-Termen). *Ist $a \in \Lambda$ S-strukturiert, definieren wir $P^s_S(a)$ mit den gleichen Definitionsgleichungen wie oben.*

4.7.4 Großzügige und kleinliche Projektionen

Für jeden Kontext C ohne `where` und jedes Kontextmuster S, das wie oben Variablen aus V_1 und V^* enthalten kann, definieren wir hier zwei weitere neue Kontexte, die „großzügige“ und die „kleinliche“ *Projektion auf die Konstruktorbestandteile* des Kontextes. Hierbei wird die Semantik benutzt, um syntaktische Teile, die keinen Termen der Konkretisierung entsprechen, zu ignorieren: Beispielsweise beinhaltet $\gamma(\texttt{c Top}\ \langle\rangle)$ keine Terme, so daß auf diesem Kontext *jede* in diesem Abschnitt definierte Projektion einen leeren Kontext zurückgibt.

Selektion

Wegen der Anwesenheit von S benötigen wir zur Vorbereitung eine *Selektion*, die Kontexte auf jene Anteile reduziert, deren Konkretisierung nur S-strukturierte Terme und auf solche reduzierende Terme enthält. Diese Funktion schneidet noch keine Stellen ab, die den Variablen x^* in S entsprechen. Vielmehr unterscheidet sie nicht zwischen x^1 und x^* und achtet nur darauf, ob ihr Argument überall in seiner Konkretisierung die Struktur von S aufweist.

Definition 4.51 (Selektion: P^0_S). *Sei S ein Kontextmuster.*
P^0_S ist eine Selektion *auf S-strukturierte Λ-Terme repräsentierende Anteile. P^0_S bildet Λ^C auf Λ^C ab und wird wie folgt definiert:*

$$P^0_S(C) := C \cap S_{\texttt{Top}}.$$

P^0_S verkleinert in der Tat einen Kontext so weit, daß von den primitiven Termen genau diejenigen, die dem Aufbau von S entsprechen, übrigbleiben:

Lemma 4.52. *Für konventionelle C gilt*

$$\begin{aligned}
\eta_{\Delta^i(\mathbf{0})}(P^0_S(C)) &= \{t \in \eta_{\Delta^i(\mathbf{0})}(C) \mid t \textit{ reduziert zu einem } S\textit{-strukturierten Term}\},\\
\eta_{\Delta^i(\mathbf{0})}(P^0_S(C)) \neq \emptyset &\iff \eta_{\Delta^i(\mathbf{0})}(C) \textit{ enthält } S\textit{-strukturierte Terme}\\
\eta(P^0_S(C)) \neq \emptyset &\iff \eta(C) \textit{ enthält } S\textit{-strukturierte Terme}
\end{aligned}$$

Beweis. Die erste Gleichung läßt sich leicht induktiv nachvollziehen. Die auftauchende Einschränkung der für die Variablen eingesetzten Terme auf $\eta_{\Delta^i(\mathbf{0})}(\texttt{Top})$ ist in Wirklichkeit keine, da für jeden konventionellen Kontext D gilt: $\eta_{\Delta^i(\mathbf{0})}(D) \subseteq \eta_{\Delta^i(\mathbf{0})}(\texttt{Top})$.
Die anderen beiden Aussagen sind Folgerungen der ersten. ◇

Großzügige und kleinliche Projektion

Wir kommen zu den oben erwähnten Projektionen. Sie unterscheiden sich in der Definition nur wenig von dem gerade erklärten P^0_S. Die „kleinliche“ Projektion nimmt einen Kontext, dessen primitive Konkretisierung nur Elemente beinhaltet, die S-strukturiert sind oder zu derartigen Ausdrücken reduzieren, und bildet ihn auf die durch die V^1-Variablen in S angegebenen Bestandteile dieser Konstrukte ab. Eine verallgemeinerte Version davon ist die „großzügige“ Projektion, welche auch andere Terme im Kontext zuläßt, sie aber ignoriert.

Definition 4.53 (Großzügige und kleinliche Kontextprojektionen: P_S, p_S, π_S). *Sei S ein Kontextmuster und C ein Kontext, der weder direkt noch indirekt* `where`*-Kontexte und Schnitte enthält, ausgenommen die Parameter von Konstruktorkontexten, die einer Variablen in S entsprechen.*
Die großzügige Projektion *P_S bildet Λ^C auf Λ^C ab und wird wie folgt definiert:*

$$\begin{aligned}
P_{x^1}(C) &:= C\\
P_{x^*}(C) &:= \langle\rangle
\end{aligned}$$

Im folgenden sei S keine Variable:

$$P_S(\langle C_1 \dots, C_n\rangle) := \langle P_S(C_1), \dots, P_S(C_n)\rangle$$

$$P_S(\mathtt{c}\ C_1 \dots C_n) := \begin{cases} (P_{S_{i_1}}(C_{i_1}), \dots, P_{S_{i_m}}(C_{i_m})) & \text{wenn } S = \mathtt{c}\ S_1 \dots S_m \\ & \text{mit } \{S_{i_1}, \dots, S_{i_m}\} = \\ & \{\, S_i \mid S_i \text{ enthält mindestens} \\ & \text{eine Variable aus } V^1 \,\} \\ & \text{und } i_1 < \dots < i_m \\ & \textbf{und } \eta(P^0_S(\mathtt{c}\ C_1 \dots C_n)) \neq \emptyset \\ \langle\rangle & \text{sonst} \end{cases}$$

$$P_S(\mathtt{Bot}) := P_S(\mathtt{Fun}) := \langle\rangle$$

$$P_S(C_1 \xrightarrow{\forall} C_2) := \begin{cases} P_S(\mathtt{Top}) & \text{wenn } \eta(C_1) = \emptyset \vee \eta(C_2) \supseteq \Lambda_p(\mathtt{Bot}) \\ \langle\rangle & \text{sonst} \end{cases}$$

$$P_S(C_1 \xrightarrow{\exists} C_2) := \begin{cases} P_S(\mathtt{Top}) & \text{wenn } \eta(C_1) \neq \emptyset \wedge \eta(C_2) \supseteq \Lambda_p(\mathtt{Bot}) \\ \langle\rangle & \text{sonst} \end{cases}$$

$$P_S(N) := N_S$$

N_S ist hierbei ein Name mit der zugehörigen neuen Kontextdefinition $N_S = P_S(C)$ bei $N = C \in \mathcal{C}$.

Die kleinliche Projektion *$p_S(C)$ sei genau dann definiert, wenn $C \leq S_{\mathtt{Top}}$. (Eine äquivalente Bedingung ist nach Lemma 4.44: $\eta(C) \subseteq \eta(S_{\mathtt{Top}})$.) Im Falle der Definiertheit sei $p_{S,\mathtt{c}}(C) := P_{S,\mathtt{c}}(C)$.*

Bezüglich eines Musters $S = (x_1, \dots, x_n)$, das nur aus Jecs und Variablen besteht, unterscheiden sich die kleinliche und großzügige Projektion im allgemeinen nicht, da Jecs für eine bestimmte Anzahl zusammen ausgewerteter Parameter stehen und diese Anzahl innerhalb eines Kontextes sinnvollerweise immer die gleiche ist; d.h. eine Jec-Konstrukte enthaltende Konkretisierung enthält nur Jec-Konstrukte, und zwar mit der jeweils gleichen Stelligkeit. Diese Projektionen von (n-stelligen) Jecs heißen π_S.

Für spezielle S verwenden wir wie bei den syntaktischen Projektionen spezielle Namen. Ganz analog seien also $P_{i,\mathtt{c}}$, $P_{\hat{\imath},\mathtt{c}}$, $p_{i,\mathtt{c}}$, $p_{\hat{\imath},\mathtt{c}}$, π_i und $\pi_{\hat{\imath}}$ definiert.

Bemerkung. *Eine Definition*

$$P_S(C_1 \cap \dots \cap C_n) := P_S(C_1) \cap \dots \cap P_S(C_n) \tag{4.11}$$

für Schnitte hätte nicht gewollte Effekte: Mit $C := (\mathtt{Bot}, \mathtt{Top}) \cap (\mathtt{[]}, \mathtt{Top})$ wäre dann $P_{(x^*, y^1)}(C) = \mathtt{Top}$, obwohl C eine leere Konkretisierung besitzt.

Es ist denkbar, daß Kontexte semantikerhaltend so transformiert werden können, daß sie ohne Schnitte auskommen; möglicherweise etwa dadurch, daß jeder Schnitt zu einer neuen Kontextdefinition wird. Dann wäre die Lücke in der obigen Definition bezüglich Schnittkontexten geschlossen. Diesem Ansatz wird jedoch im Rahmen dieser Arbeit nicht weiter nachgegangen.

Das folgende zentrale Lemma stellt den Zusammenhang zwischen der definierten Projektion und deren gewünschter Semantik her: Die primitiven Terme von Projektionen entsprechen syntaktischen Projektionen der primitiven Terme des ursprünglichen Kontextes.

Lemma 4.54. *Für konventionelle C, auf denen $P_S(C)$ definiert ist, gilt*

$$\eta(P_S(C)) = \{\, P^s_S(t) \mid t \in \eta(C) \text{ und } t \text{ ist } S\text{-strukturiert} \}$$

Beweis. Seien

$$M_j(C) := \{\, P^s_S(t) \mid t \in \eta_{\Delta^j(\mathbf{0})}(C) \text{ und } t \text{ ist } S\text{-strukturiert} \} \text{ und}$$
$$M(C) := \{\, P^s_S(t) \mid t \in \eta(C) \text{ und } t \text{ ist } S\text{-strukturiert} \}$$

Wir zeigen, daß für alle $j \geq 0$ und $C \in \mathcal{C}$ gilt:[3]

$$M_j(C) \subseteq \eta_{\Delta^j(\mathbf{0})}(P_S(C)) \subseteq M(C).$$

Die Vereinigung über alle j ergibt dann die Aussage des Lemmas.

Der Beweis ist eine Induktion über die Struktur von C und über j, die wieder mit der Induktionsordnung $(j, C) < (j', C') \iff j < j' \vee (j = j' \wedge C$ ist echter Teilkontext von $C')$ funktioniert:

- $S = x$: $\eta_{\Delta^j(\mathbf{0})}(C) \subseteq \eta_{\Delta^j(\mathbf{0})}(C) \subseteq \eta(C)$.
- $S = x^*$: $\emptyset \subseteq \emptyset \subseteq \emptyset$.

Im folgenden sei S keine Variable.

- $C = N$:
  - $j = 0$: $\emptyset \subseteq \emptyset \subseteq M(C)$.
  - $j > 0$: Sei $N = D \in \mathcal{C}$. Es ist
    $$M_j(N) = M_{j-1}(D) \overset{IV}{\subseteq} \begin{array}{l}\eta_{\Delta^{j-1}(\mathbf{0})}(P_S(D)) \\ = \eta_{\Delta^j(\mathbf{0})}(P_S(N))\end{array} \overset{IV}{\subseteq} M(D) = M(N).$$
- $C = \mathtt{c}\ C_1 \ldots C_m$:
  $$\{\, P^s_S(t) \mid t \text{ ist } S\text{-strukturiert} \wedge t \in \eta_{\Delta^j(\mathbf{0})}(\mathtt{c}\ C_1 \ldots C_n) \,\} = M_j(C)$$
  $$= \begin{cases} \{\, (P^s_{S_{i_1}}(t_{i_1}), \ldots, P^s_{S_{i_m}}(t_{i_m})) \mid & \text{wenn } S = \mathtt{c}\ S_1 \ldots S_n \\ \forall \lambda : t_{i_\lambda} \in \eta_{\Delta^j(\mathbf{0})}(C_{i_\lambda}) & \wedge \eta_{\Delta^j(\mathbf{0})}(P^0_S(C)) \neq \emptyset \\ \wedge\, t_{i_\lambda} \text{ ist } S_{i_\lambda}\text{-strukturiert} \,\} & \\ \emptyset & \text{sonst} \end{cases}$$
  $$\overset{IV}{\subseteq} \begin{cases} \eta_{\Delta^j(\mathbf{0})}((P_{S_{i_1}}(C_{S_{i_1}}), \ldots, P_{S_{i_m}}(C_{S_{i_m}}))) & \\ \qquad \text{wenn } S = \mathtt{c}\ S_1 \ldots S_n \wedge \eta_{\Delta^j(\mathbf{0})}(P^0_S(C)) \neq \emptyset & \\ \emptyset \qquad \text{sonst} & \end{cases}$$
  $$\subseteq \left\{ \begin{array}{ll} \eta_{\Delta^j(\mathbf{0})}((P_{S_{i_1}}(C_{S_{i_1}}), \ldots, P_{S_{i_m}}(C_{S_{i_m}}))) & \\ \qquad \text{wenn } S = \mathtt{c}\ S_1 \ldots S_n \wedge \eta(P^0_S(C)) \neq \emptyset & \\ \emptyset \qquad \text{sonst} & \end{array} \right\} = \eta_{\Delta^j(\mathbf{0})}(P_S(C))$$
  $$\overset{IV}{\subseteq} \begin{cases} \{\, (P^s_{S_{i_1}}(t_{i_1}), \ldots, P^s_{S_{i_m}}(t_{i_m})) \mid & \text{wenn } S = \mathtt{c}\ S_1 \ldots S_n \\ \forall \lambda : t_{i_\lambda} \in \eta(C_{i_\lambda}) & \wedge \eta(P^0_S(C)) \neq \emptyset \\ \wedge\, t_{i_\lambda} \text{ ist } S_{i_\lambda}\text{-strukturiert} \,\} & \\ \emptyset & \text{sonst} \end{cases}$$
  $$= \{\, P^s_S(t) \mid t \text{ ist } S\text{-strukturiert} \wedge t \in \eta(\mathtt{c}\ C_1 \ldots C_n) \,\} = M(C)$$
- $C = \langle C_1, \ldots, C_m \rangle$:
  $$M_j(C) = \textstyle\bigcup_k M_j(C_k) \overset{IV}{\subseteq} \begin{array}{l}\bigcup_k \eta_{\Delta^j(\mathbf{0})}(P_S(C_k)) \\ = \eta_{\Delta^j(\mathbf{0})}(P_S(C))\end{array} \overset{IV}{\subseteq} \textstyle\bigcup_k M(C_k) = M(C)$$
- $C \in \{\mathtt{Bot}, \mathtt{Fun}\}$: $\emptyset \subseteq \emptyset \subseteq \emptyset$.

$\diamond$

[3] Diese Aussage läßt sich vermutlich auch für nicht-konventionelle Kontexte beweisen, im Rahmen des Lemmas wird das aber nicht benötigt.

Man kann den bei der Definition von $P_S(\mathtt{c}\ldots)$ gebrauchten Test $\eta(P^0_S(\mathtt{c}\ C_1 \ldots C_n)) \neq \emptyset$ durch einen anderen ersetzen, bei dem weniger Kontexte vorkommen. Er ist deshalb möglicherweise schneller zu berechnen und wird weiter unten auch für `where`-Kontexte benötigt.

Lemma 4.55. *Die Bedingung im Konstruktorfall bei $S = \mathtt{c}\ S_1 \ldots S_n$, $\eta(P^0_S(\mathtt{c}\ C_1 \ldots C_n)) \neq \emptyset$, läßt sich ersetzen durch die folgende Bedingung*

$$\eta(P^0_{(S_{k_1},\ldots,S_{k_r})}((C_{k_1},\ldots,C_{k_r}))) \neq \emptyset$$
$$\textit{mit } \{S_{k_1},\ldots,S_{k_r}\} = \{S_k | S_k \textit{ enthält keine } \textit{Variable aus } V^1\},$$

ohne daß sich die Konkretisierung des definierten Kontextes ändert.

Beweis. Der einzige Fall, in dem die Resultate unterschiedlich werden können, ist dann gegeben, wenn $S = \mathtt{c}\ S_1 \ldots S_n$ ist und $\eta(P^0_{(S_{k_1},\ldots,S_{k_r})}((C_{k_1}\ldots C_{k_r}))) \neq \emptyset$, aber $\eta(P^0_S(\mathtt{c}\ C_1 \ldots C_n)) = \emptyset$ gilt. Die neue Definition gibt $(P_{S_{i_1}}(C_{i_1}),\ldots,P_{S_{im}}(C_{im}))$ zurück, die ursprüngliche Definition $\langle\rangle$. Wir argumentieren, daß der erste der beiden Terme wie der zweite eine leere Konkretisierung besitzt:

$P^0_S(\mathtt{c}\ldots)$ und $P^0_{(S_{k_1},\ldots,S_{k_r})}((\ldots))$ sind über die $P^0_{S_j}(C_j)$ definiert. Die Voraussetzungen des Falles implizieren also, daß es ein $\eta(P^0_{S_j}(C_j))$ gibt, welches verschwindet, aber kein $\eta(P^0_{S_{k_j}}(C_{k_j}))$ leer ist. Somit muß für eines der restlichen Konstruktorargumente, das sind gerade die C_{i_1} bis C_{im} aus der Definition von P_S, gelten: $\eta(P^0_{S_{i_j}}(C_{i_j})) = \emptyset$. Das bedeutet aber nach Lemma 4.52, daß $\eta(C_{i_j})$ keine S_{i_j}-strukturierten Terme enthält.

Nach dem eben bewiesenen Lemma 4.54 muß dann auch $\eta(P^s(C_{i_j}))$ leer sein, so daß auch der ganze Jec $(P_{S_{i_1}}(C_{i_1}),\ldots,P_{S_{im}}(C_{im}))$ keine Λ-Werte repräsentiert. ◇

Großzügige Projektion auf Λ-Termen

Um eine bequeme Ausdrucksweise bei Λ-Ausdrücken zu haben, sollen P_S und alle Spezialfälle davon, wie bei der syntaktischen Projektion, auch Namen für auf Λ-Teilmengen definierte Projektionen mit analogen Eigenschaften sein. So wie man P_S als Erweiterung von P^s_S auf allgemeine Kontexte ansehen kann, ist P_S auf Λ-Teilmengen die Fortsetzung von P^s_S auf Λ-Teilmengen:

Definition 4.56 (P_S, π_S auf Λ-Termen). *P_S sei nicht nur die oben definierten Kontextprojektion, sondern stehe zusätzlich für eine Abbildung, die $\mathcal{P}(\Lambda)$ auf $\mathcal{P}(\Lambda)$ abbildet mit*

$$P_S(M) = \{\, P^s_S(t) \,|\, t \in M \textit{ und } t \textit{ ist } S\textit{-strukturiert} \}$$

Zwar sind die Symbole dadurch mit verschiedenen Bedeutungsebenen mehrfach belegt, doch der Zusammenhang macht den Sinn immer deutlich. Die Aussage von Lemma 4.54 kann man jetzt noch kompakter darstellen:

$$\eta(P_S(C)) = P_S(\eta(C)).$$

Vor allem π_i und $\pi_{\bar{i}}$ werden in dem eben definierten Sinne öfter zur Verwendung kommen. Man bemerke, daß für konventionelle C z.B.

$$\eta(\pi_i(C)) = \pi_i(\eta(C)) = \{\, t \,|\, \exists a \in \eta(C) : a = (a_1,\ldots,a_n) \wedge t = a_i \,\}.$$

4.7.5 Projektionen auf `where`-Kontexten

Die Projektion $P_S(R\ \mathtt{where}\ T = N)$ wird eventuell selbst wieder ein `where`-Ausdruck sein. Bei den nachfolgend gegebenen Definitionen wird darauf geachtet, `where`-Ausdrücke $R'\ \mathtt{where}\ T' = N'$ nur in einer bestimmten Form zu erzeugen, nämlich so, daß $\mathcal{FV}(R') = \mathcal{FV}(T')$. Diese Form mußten `where`-Ausdrücke in einer früheren Version der Kontextanalyse aufweisen. Es hat sich erst während

der Entstehung dieser Diplomarbeit herausgestellt, daß aufgrund veränderter Beweistechniken bei der Kontextanalyse diese Einschränkung nicht mehr nötig ist.

So definieren wir $P_S(R \texttt{ where } T = N)$ für einige Spezialfälle. Es ist denkbar, Projektionen statt dessen auch allgemein auf `where`-Kontexten zu erklären, wenn man die oben genannte Einschränkung der Kontextsyntax wieder fallen läßt. $P_S(R \texttt{ where } T = N)$ könnte vermutlich als $P_S^{T,N}(R) \texttt{ where } T = N$ definiert werden, wobei $P_S^{T,N}$ eine ähnliche Projektion ist, die T und N berücksichtigt. Man müßte sich dann aber zusätzlich überlegen, wie man Projektionen auf freien Variablen definiert; also was passiert, wenn eine Projektion in eine Variable „hineingreifen" muß.

Zwei der nun zu definierenden Spezialfälle sind recht kompliziert, so daß sie im Anhang aufgeführt werden. Die restlichen Fälle sind relativ einfach gestrickt:

Definition 4.57. *1. Falls $S = x$ oder $S = x^*$:*

$P_S(R \texttt{ where } T = N) := R \texttt{ where } T = N$ *bzw.* $\langle\rangle$.

Im folgenden sei S keine Variable.

2. Falls $S \neq \texttt{c } S_1 \dots S_n$:

$P_S(\texttt{c } C_1 \dots C_n \texttt{ where } T = N) := \langle\rangle$

Im folgenden sei $S = \texttt{c } S_1 \dots S_n$. Die S_i, in denen eine Variable aus V^1 vorkommt, seien $S_{i_1}, \dots, S_{i_m}$, und die restlichen S_k, in denen also keine Variable aus V^1 vorkommt, seien $S_{k_1}, \dots, S_{k_r}$.

3. Die S_{i_j} seien selbst Variablen (enthalten also keine Konstruktoren). Falls in den Komponenten $C_{k_1}, \dots, C_{k_r}$ von R, an deren entsprechender Stelle in S sich keine Variablen aus V^1 befinden, keine freien Variablen vorkommen:

$$P_{(\texttt{c } S_1 \dots S_n)}(\texttt{c } C_1 \dots C_n \texttt{ where } T = N) := \begin{cases} (C_{i_1}, \dots, C_{i_m}) \texttt{ where } T = N & \text{wenn } \eta(P^0_{(S_{k_1}, \dots, S_{k_r})}((C_{k_1}, \dots, C_{k_r}))) \neq \emptyset \\ \langle\rangle & \text{sonst} \end{cases}$$

4. Die S_{k_j} seien selbst Variablen (enthalten also keine Konstruktoren). Falls in den Komponenten $C_{i_1}, \dots, C_{i_m}$ von R, an deren entsprechender Stelle in S sich Variable(n) aus V^1 befinden, keine freien Variablen vorkommen:

$$P_{(\texttt{c } S_1 \dots S_n)}(\texttt{c } C_1 \dots C_n \texttt{ where } T = N) := \begin{cases} P_{(S_{i_1}, \dots, S_{i_m})}(C_{i_1}, \dots, C_{i_m}) & \text{wenn } \eta((C_{k_1}, \dots, C_{k_r}) \texttt{ where } T = N) \neq \emptyset \\ \langle\rangle & \text{sonst} \end{cases}$$

Es fällt auf, daß die Definition im Fall 3 den Test auf $\eta(P^0_{(S_{k_1}, \dots, S_{k_r})}((C_{k_1} \dots C_{k_r}))) \neq \emptyset$ benutzt, im Fall 4 aber vorausgesetzt wird, daß die S_{k_j} Variablen sind, so daß auf $\eta((C_{k_1} \dots C_{k_r}) \texttt{ where } T = N) \neq \emptyset$ getestet wird, also der Fall 4 an dieser Stelle weniger allgemein ist als Fall 3. Wenn der Test erfolgreich ist, wird jedoch im Fall 4 der Kontext $P_{(S_{i_1}, \dots, S_{i_m})}(C_{i_1}, \dots, C_{i_m})$ erzeugt, während unter Punkt 3 an dieser Stelle $(C_{i_1}, \dots, C_{i_m}) \texttt{ where } T = N$ steht ohne Erwähnung von $P_{(S_{i_1}, \dots, S_{i_m})}$, weil dort wiederum vorausgesetzt ist, daß die S_{i_j} Variablen sind.

Der Grund dieser Spezialisierung liegt darin, daß die Kontexte $(C_{i_1}, \dots, C_{j_m})$ und $(C_{k_1}, \dots, C_{k_r})$, um die es sich handelt, gerade an den bewußten Stellen Teile von `where`-Kontexten sind; sie enthalten noch Variablen. Wollte man Projektionen auch auf Kontexten mit freien Variablen definieren, so wären mehrere Schwierigkeiten zu beachten:

- Die Projektion könnte aus einem Ausdruck Teile entfernen, die Variablen enthalten; diese Variablen fehlten dann im projizierten Kontext, so daß sie aufgrund der weiter oben genannten Syntaxbeschränkung auch aus der Quantifizierung „$\texttt{where } T = N$" entfernt werden müßten.[4]

[4] Man könnte dafür allerdings das Absorptionsgesetz der Mengenlehre ausnutzen und z.B. $\langle \texttt{x} \cap C, C\rangle \texttt{ where } T = N$ schreiben, wenn `x` nicht in C vorkommt, aber in T. Das sieht aber sehr unschön aus und bietet keine sehr konstruktiven Informationen.

- Auch der Versuch einer Projektion, ins Innere einer einzelnen Variable „hineinzugreifen", müßte behandelt werden.

Für diese Probleme eine Lösung zu finden, die einer Korrektheitsüberprüfung standhält, würde einen für diese Arbeit unverhältnismäßig großen Aufwand darstellen.

Wir beschränken uns daher auf die oben definierten Spezialfälle. Nun beweisen wir, daß auch die `where`-Projektionen die gewünschten primitiven Terme besitzen:

Lemma 4.58. *Lemma 4.54 gilt auch für Kontexte C, die Kontexte der oben angegebenen Formen enthalten.*

Beweis. Wir benutzen die Terminologie aus Lemma 4.54 und können die Induktionsvoraussetzung anwenden:

$$\begin{aligned} M_j(R \texttt{ where } T = N) &= \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta_{\Delta^j(\mathbf{0})}(N)} M_j(\sigma R) \\ &\overset{IV}{\subseteq} \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta_{\Delta^j(\mathbf{0})}(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) \\ &\subseteq \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) \\ &\overset{IV}{\subseteq} \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} M(\sigma R) = M(R \texttt{ where } T = N) \end{aligned}$$

Wir müssen also folgendes zeigen:

$$\bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta_{\Delta^j(\mathbf{0})}(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) \overset{!}{\subseteq} \eta_{\Delta^j(\mathbf{0})}(P_S(R \texttt{ where } T = N)) \overset{!}{\subseteq} \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R))$$

für jeden der oben angeführten Fälle. Meistens wird dabei Gebrauch von Lemma 4.55 gemacht werden.

1. Hier herrscht Gleichheit zwischen dem linken und dem mittleren Ausdruck: Es ist $\bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta_{\Delta^j(\mathbf{0})}(N)} \eta_{\Delta^j(\mathbf{0})}(\sigma R) = \eta_{\Delta^j(\mathbf{0})}(R \texttt{ where } T = N)$ bzw. $\emptyset = \emptyset$. Die zweite Inklusion muß deshalb nicht mehr gezeigt werden.

2. $\emptyset \subseteq \emptyset \subseteq \emptyset$.

3. Auch hier sind der linke und der mittlere Term gleich:

$$\begin{aligned} &\bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta_{\Delta^j(\mathbf{0})}(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) \\ &= \begin{cases} \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta_{\Delta^j(\mathbf{0})}(N)} \eta_{\Delta^j(\mathbf{0})}(\sigma(C_{i_1},\ldots,C_{i_m})) & \text{wenn } \eta(P^0_{(S_{k_1},\ldots,S_{k_r})}(\underbrace{(C_{k_1},\ldots,C_{k_r})}_{\substack{=\sigma(C_{k_1},\ldots,C_{k_r})\\ \text{für alle } \sigma}})) \neq \emptyset \\ \emptyset & \text{sonst} \end{cases} \\ &= \eta_{\Delta^j(\mathbf{0})}(P_S(R \texttt{ where } T = N)) \end{aligned}$$

gilt unter Benutzung von Lemma 4.55.

4. Hier ist der mittlere Ausdruck gleich dem rechten.

$$
\begin{aligned}
&\bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) \\
&= \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} \begin{cases} \eta_{\Delta^j(\mathbf{0})}(P_{(S_{i_1},\ldots,S_{i_m})}(\underbrace{\sigma(C_{i_1},\ldots,C_{i_m})}_{=(C_{i_1},\ldots,C_{i_m})})) & \text{wenn } \eta(\sigma(C_{k_1},\ldots,C_{k_r}))\neq\emptyset \\ \emptyset & \text{sonst} \end{cases} \\
&= \begin{cases} \eta_{\Delta^j(\mathbf{0})}(P_{(S_{i_1},\ldots,S_{i_m})}((C_{i_1},\ldots,C_{i_m}))) & \text{wenn } \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} \eta(\sigma(C_{k_1},\ldots,C_{k_r}))\neq\emptyset \\ \emptyset & \text{sonst} \end{cases} \\
&= \eta_{\Delta^j(\mathbf{0})}(P_S(R \texttt{ where } T = N))
\end{aligned}
$$

◇

4.8 Komplemente

Wir gehen davon aus, daß es zu jedem Kontext C, der weder direkt noch indirekt über Namensbezüge `where`-Ausdrücke enthält, einen Komplementkontext C^c gibt mit

$$\eta(C^c) = \Lambda_p \setminus \eta(C).$$

In [SSS97b] findet sich ein Kalkül, der ein C^c berechnet, das recht nahe an diese Eigenschaft herankommt. Er weist jedoch eine bedeutende Lücke auf: Z.B. gibt er für einen Kontext L mit der Definition $L = L$ das Ergebnis L^c aus mit der Definition $L^c = L^c$, was also die gleiche (leere) Konkretisierung hat. Ein anderes Beispiel sei M mit $M = \langle M, \texttt{Inf}\rangle$; es ist $\eta(M) = \eta(\texttt{Inf})$. Der Kalkül berechnet ein M^c mit der Definition $M^c = M^c \cap \texttt{Inf}^c$; dabei ist $\eta(M^c) = \emptyset$.

Sei ζ eine Abbildung, die wie η definiert ist, aber über den *größten* statt über den kleinsten Fixpunkt. (Ein Beweis wie für Lemma 4.22 sichert dessen Existenz.) Wir vermuten folgendes:

1. Der Kalkül aus [SSS97b] erzeugt für C ein C^c mit $\zeta(C^c) = \Lambda_p \setminus \eta(C)$. (Dies ist bei den obigen Beispielen erfüllt.)

2. Für einen Kontext D ohne Bezug auf `where`-Ausdrücke gilt $\eta(D) = \zeta(D)$, wenn seine Definition keine Toplevel-Rekursionen enthält; damit ist folgendes gemeint:

   Sei N ein Kontextname mit Definition $N = X$, auf den sich D direkt oder indirekt bezieht. Dann dürfen Namen, die sich direkt oder indirekt auf N beziehen, nur dann in X auftauchen, wenn sie sich dort im Argument eines Konstruktorkontextes befinden.

3. Eine semantikerhaltende Modifikation von $\mathcal{C}$, bei der jede Namensdefinition frei von Toplevel-Rekursionen ist, ist möglich. Dies funktioniert bei jedem Namen mittels fortgesetzter Definitionsexpansion oder durch Ersetzen von Namen durch den leeren Kontext. (Im obigen Beispiel würde man $L = \langle\rangle$ und $M = \langle\langle\rangle, \texttt{Inf}\rangle$ definieren.)

Aus der Richtigkeit dieser Vermutungen würde folgen, daß das gewünschte Komplement dann erzeugt wird, wenn der Kalkül aus [SSS97b] auf dem modifizierten $\mathcal{C}$ gestartet wird.

Zu zeigen, ob die Vermutungen richtig sind, führt aus dem Rahmen dieser Arbeit. Sollten sie nicht gelten, können Komplemente immer noch approximiert werden.

Die Komplemente der Kontexte $C_1 \overset{\forall}{\to} C_2$ und $C_1 \overset{\exists}{\to} C_2$ sind $C_1^c \overset{\exists}{\to} C_2$ bzw. $C_1^c \overset{\forall}{\to} C_2$, wie aus den Definitionen 4.28 und 4.35 ersichtlich wird. Wenn C_1^c $\overset{\forall}{\to}$-Ausdrücke enthält, entstünde so allerdings ein nicht zugelassener Kontext $C_1^c \overset{\forall}{\to} C_2$. Diesen Fall wollen wir deshalb in dieser Arbeit ausschließen.

Kapitel 5

Kalkül

In diesem Kapitel stellen wir die *Kontextanalyse* aus [SSS97a] und [Sch99b] vor, die in ihrem Kern ein Tableaukalkül ist. Sie wird im letzten Abschnitt mit Regeln erweitert, die sich für Argumente höherer Ordnung eignen.

Bei der Darstellung der Kontextanalyse folgen wir [Sch99b]. In dieser Quelle wird auch der Kalkül derzeit noch weiterentwickelt; wir können also nur einen Stand der Entwicklung wiedergeben.

5.1 Kontextanalyse

Die *Kontextanalyse* aus [SSS97a] ist ein Verfahren, das für nicht-strikte funktionale Sprachen Kontextinformationen errechnet, indem es Anfragen auf einem *Tableau* mittels abstrakter Reduktion bearbeitet.

Es handelt sich um eine Verallgemeinerung der Striktheitsanalyse: Diese versucht, strikte Argumente von Funktionen zu finden, also solche Argumente, die immer ausgewertet werden. Das heißt, die Nichtterminierung dieser Argumente zieht auch die Nichtterminierung der ganzen Funktionsanwendung nach sich.

Dagegen versucht die Kontextanalyse, die Werte der Argumente, die z.B. ein nichtterminierendes Ergebnis erzeugen, zu *konstruieren*. Statt der Ja/Nein-Antworten der Striktheitsanalyse sind also genauere Informationen möglich. Allgemeiner noch: Es kann ein beliebiger Ergebniskontext vorgegeben werden, und es werden Werte (d.h. Kontexte) für die Argumente konstruiert, für die das Ergebnis der Funktionsanwendung von dem vorgegebenen Kontext repräsentiert wird. Eine solche Analyse ist für viele Zwecke nutzbar (s. [Sch99b]), unter anderem auch als Werkzeug für Programmtests, wie in der Einleitung schon dargelegt wurde.

Als Tableaukalkül steht die Kontextanalyse in der Tradition anderer Analysen auf funktionalen Programmen, die schon mit dieser Technik erfolgreich waren. Hier sind die Striktheitsanalyse ([Sch94],[SSPS95]) und die Terminierungsanalyse ([Pan96],[Kic96]) zu erwähnen. Dabei wird versucht, die gestellte Anfrage auf leichtere Formeln zu reduzieren, wobei die zurückgelegten Schritte die Form eines Baums, nämlich eines Tableaus, bilden; eine Methode, die aus dem Gebiet der Logik stammt.

Wir beschreiben nun die Kontextanalyse: Dem Kalkül wird eine Anfrage der Form $f \in C$ mit $f \in \Lambda$, $C \in \Lambda^C$, gestellt. $x_1, \ldots, x_n$ seien die freien Variablen in f; üblicherweise ist f die Anwendung eines Superkombinators auf eine oder mehrere Variablen. Es soll nun ein Kontext D als Ergebnis geliefert werden, so daß für $(a_1, \ldots, a_n) \in \gamma(D)$ folgt: $f(a_1, \ldots, a_n) \in \gamma(C)$.

Die eben benutzte Notation wird oft verwendet werden:

Schreibweise. Sind $x_1, \ldots, x_n$ in einem Ausdruck f vorkommende Variablen, so sei mit $f(\cdot, \ldots, \cdot)$ ein n-stelliger syntaktischer Mehrloch-Kontext erklärt, so daß $f(x_1, \ldots, x_n) \equiv f$.

Das Beispiel in Abbildung 5.1 wird schon vor der nun anstehenden Formalisierung des Kalküls einen Eindruck von dessen Vorgehensweise geben. Dort sind die Substitutionen auf den Kanten des Tableaus aufgeführt statt in den Knoten; dies veranschaulicht deren akkumulierenden Aufbau durch die Regeln. (Substitutionen und Regeln werden weiter unten definiert.) Die benutzten Superkombinatoren sind durch

$\texttt{append}\ xs\ ys = \texttt{case}_{List}\ xs\ ys\ (\texttt{appc}\ ys)$
$\texttt{appc}\ ys\ z\ zs\ = z\texttt{:}(\texttt{append}\ zs\ ys)$

definiert.

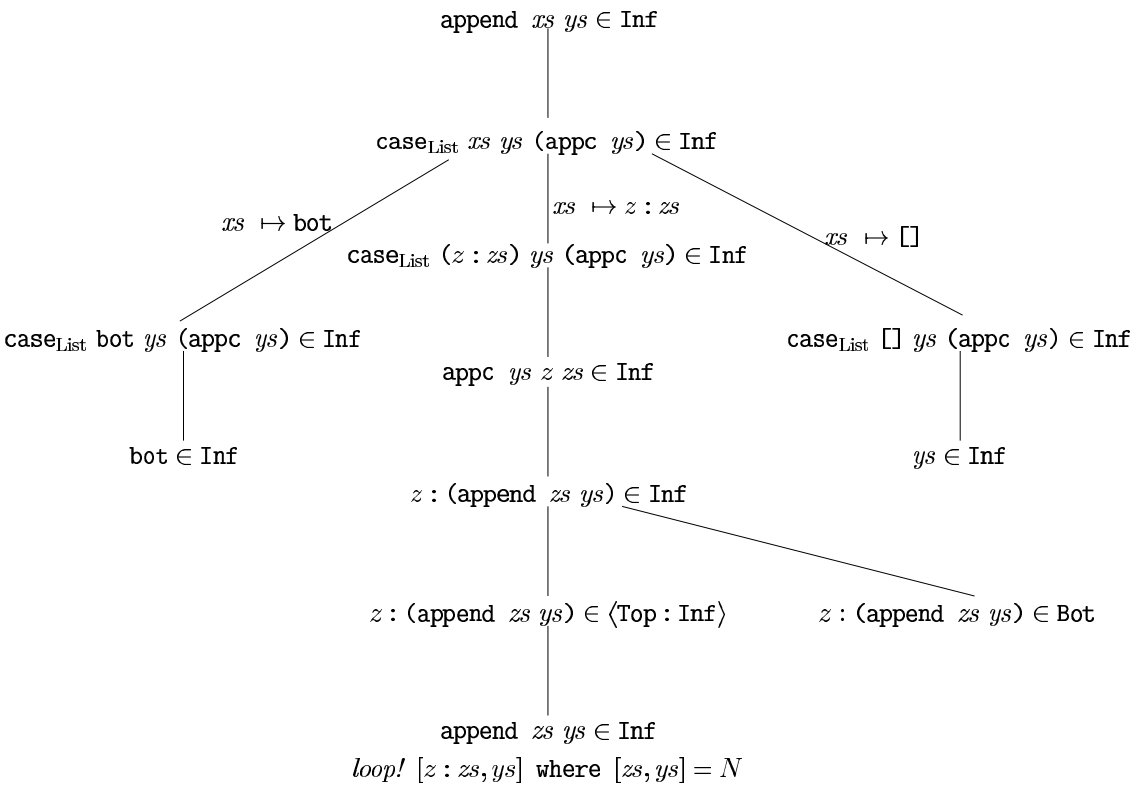

Abbildung 5.1: Kontextanalyse auf `append xs ys ∈ Inf`

5.1.1 Tableaus

Der entstehende Baum wird als *Tableau* bezeichnet.

Definition 5.1 (Tableau). *Ein* Tableau *ist ein endlicher Baum, dessen Knoten eine Marke der Form* (rs, σ) *tragen.* rs *ist eine Menge von Bedingungen oder* Constraints *der Form* $s \in C$ *mit* $s \in \Lambda$ *und* $C \in \Lambda^C$. σ *ist eine* Substitution *über* Λ, *d.h. eine Variablen ersetzende und die Struktur von Ausdrücken respektierende Abbildung aus* $\Lambda \to \Lambda$, *und ersetzt Variablen hier durch Terme aus* `bot`, *Konstruktoren und Variablen.*

σ wird dabei gewöhnlich als eine Menge von einzelnen Abbildungen der Form $x_i \mapsto t_i$ geschrieben für Variablen x_i. Die Constraints eines Knotens schreibt man auch $s \in A, R$, wenn man Bezug nimmt auf eine bestimmte Bedingung $s \in A$ und die (möglicherweise nicht vorhandenen) restlichen Constraints R. Ist s eine Variable, so wird ein Constraint $s \in C$ ein *CCV* genannt (englisch: context constraint for variables).

Die Wurzel trägt die identische Substitution sowie üblicherweise Constraints der Form $s \in A, R$, wobei s wohlgetypt ist, A weder `where`-Ausdrücke enthält noch sich über Kontextnamen auf solche

bezieht und R eine Menge von CCVs ist. Für die freien Variablen der Wurzel wird eine feste Ordnung angenommen, so daß sich die Substitutionen in den Knoten auch als Tupel schreiben lassen.

In den Blättern können zusätzliche Marken stehen: *no!* für Blätter ohne Lösung und *loop!* w für Schleifen; dabei ist w ein `where`-Ausdruck.

Definition 5.2 (geschlossenes Tableau). *Ein Tableau ist* geschlossen, *wenn jedes Blatt*

- *mit* no! *markiert ist oder*
- *mit* loop! w *markiert ist oder*
- gelöst *ist, d.h. alle seine Constraints CCVs sind.*

Ein geschlossenes Tableau ist eine Darstellung für einen Kontext, nämlich für einen Jec für die freien Variablen der Wurzel. Eine Kontextdefinition für das Tableau wird erzeugt, indem man einen eindeutigen Namen vergibt und eine Vereinigung der Kontextinformationen aller nicht mit *no!* markierten Blättern erzeugt: Für Variablen, die entweder in einem CCV erscheinen oder von der Substitution behandelt werden, läßt sich der zugehörige Kontext leicht ablesen; trifft beides auf eine Variable zu, bildet man den Schnitt der jeweiligen Kontexte. Eine Variable x, die in einem Blatt nicht vorkommt, kann betrachtet werden als durch $x \in \texttt{Top}$ beschränkt. Die so für die freien Variablen an der Wurzel gefundenen Kontexte werden pro Blatt in einem Jec zusammengefaßt.

Jeder Knoten repräsentiert eine Menge von *Lösungen*, ebenso das gesamte Tableau:

Definition 5.3 (Lösung). *θ sei eine Substitution über Λ, die im Gegensatz zu den in den Tableauknoten befindlichen Substitutionen beliebige Λ-Terme für die Variablen einsetzen darf.*

θ ist eine Lösung des Knotens (rs, σ) *mit* $rs = \{s_1 \in C_1, \ldots, s_n \in C_n\}$ *genau dann, wenn*

$$\forall i : \theta s_i \in \gamma(C_i).$$

Eine solche Lösung ist Lösung der Wurzel (ts, ϱ), $ts = \{t_i \in D_i\}$ *und damit des Tableaus, wenn*

$$\forall i : \theta\sigma t_i \in \gamma(D_i).$$

Die Menge der Lösungen von (rs, σ) wird mit $U(rs, \sigma)$ bezeichnet.

Das Tableau wird ausgehend von der Wurzel mittels Regeln expandiert, die im nächsten Abschnitt beschrieben werden. Dabei ist eine korrekte Expansion erwünscht, so daß nur Lösungen berechnet werden, die die Anfrage an der Wurzel auch wirklich erfüllen. Der Preis dafür ist, daß Vollständigkeit nicht immer gegeben sein wird.

Definition 5.4 (korrektes Tableau, vollständiges Tableau). *Ein Tableau ist* korrekt, *wenn für alle inneren Knoten $N = (rs, \sigma)$ mit Kindern $K_i = (ts_i, \pi_i \circ \sigma)$, $i = 1, \ldots, m$ gilt:*

$$\bigcup_i \pi_i U(N_i) \subseteq U(N).$$

Ein Tableau ist vollständig, *wenn für alle inneren Knoten N mit Kindern K_i und den Bezeichnungen wie eben gilt:*

$$U(N) \subseteq \bigcup_i \pi_i U(N_i).$$

Dazu ist noch zu sagen, daß die Substitution θ eines Kindknotens K von $N = (rs, \sigma)$ immer als $\theta = \pi \circ \sigma$ geschrieben werden kann. Alle der im folgenden beschriebenen Regeln, die eine neue Substitution erzeugen, führen diese nämlich nur als Komposition mit der des Elternknotens ein.

5.1.2 Regeln

Die Entwicklung des Tableaus findet durch transformierende Regeln statt. Es handelt sich dabei um *Expansionsregeln*, die ein Tableau jeweils an einem Blatt dadurch erweitern, daß sie neue Blätter an die alten hängen.

Definition 5.5 (korrekte Regel, vollständige Regel). *Eine Regel ist korrekt, wenn sie korrekte Tableaus in korrekte Tableaus verwandelt. Eine Regel ist vollständig, wenn sie vollständige Tableaus in vollständige Tableaus verwandelt.*

Wir erwähnen kurz die in der Kontextanalyse angewendeten Regeln. Wenn hier gesagt wird, daß z.B. ein Constraint R gestrichen werden kann, so ist damit gemeint, daß ein Blatt angehängt wird, das durch die Entfernung von R aus seinem Elternknoten entsteht.

Einige Regeln sind in Gentzen-Notation geschrieben: $\frac{A}{A'}$, wobei A ein Blatt ist und A' das daran neu angehängte Blatt. Das Anhängen von mehreren Blättern wird $\frac{A}{A_1|\cdots|A_m}$ notiert.

Regeln für `Bot` Ein Constraint $\texttt{bot} \in \texttt{Bot}$ kann gestrichen werden.

Enthält ein Blatt ein Constraint $\texttt{bot} \in A$ und ist $\texttt{Bot} \not\leq A$, so kann es mit *no!* markiert werden.

Vereinigungen Ein Vereinigungskontext kann zerlegt werden:

$$\frac{s \in \langle C_1, \ldots, C_m \rangle, R, \sigma}{s \in C_1, R, \sigma | \cdots | s \in C_m, R, \sigma}$$

Schnitte Muß ein Term mehreren Bedingungen genügen, so können diese zusammengefaßt werden:

$$\frac{s \in C, s \in D, R, \sigma}{s \in C \cap D, R, \sigma}$$

Diese Regel kann auch umgekehrt werden, ist aber dann möglicherweise nicht korrekt:

$$\frac{s \in C \cap D, R, \sigma}{s \in C, s \in D, R, \sigma}$$

Die beiden Regeln heben den Effekt der jeweils anderen auf. Es bleibt der Implementierung überlassen, damit umzugehen.

no-Reduktion Gilt $s \rightarrow_{no} t$, so kann ein Constraint $s \in C$ durch $t \in C$ ersetzt werden.

Typfehler Enthält ein Blatt ein Constraint $s \in C$ mit $s \in \mathrm{NWT}$, so wird es mit *no!* markiert, da wir mit einer wohlgetypten Wurzel starten. Diese Regel wird nicht in allen Fällen Vollständigkeit bewahren, da für $\texttt{Bot} \leq C$ auch nichtwohlgetypte s akzeptiert werden könnten.

Konstruktoren Sind in einem Constraint $s \in C$ sowohl s als auch C Konstruktorterme, die mit übereinstimmenden Konstruktoren aufgebaut sind, kann mit den jeweiligen Teiltermen fortgefahren werden („Konstruktordekomposition“):

$$\frac{\texttt{c}\ t_1 \ldots t_n \in \texttt{c}\ C_1 \ldots C_n, R, \sigma}{t_1 \in C_1, \ldots, t_n \in C_n, R, \sigma}$$

Enthält ein Blatt ein Constraint $\texttt{c}\ t_1 \ldots t_n \in A$ und ist $\gamma(A \cap \texttt{c}\ \texttt{Top} \ldots \texttt{Top}) = \emptyset$, so kann es mit *no!* markiert werden.

Regel für `case` Ein $\texttt{case}_A$, welches auf einen Ausdruck in SCWHNF angewendet wird, kann mit der obigen Reduktionsregel behandelt werden.

Ist das Argument des $\texttt{case}_A$ eine Variable x, instanziieren wir wie folgt die verschiedenen Möglichkeiten, wie x aussehen kann. Dabei werden im Konstruktorfall neue Variablen eingeführt.

Seien $D[\cdot]$ ein syntaktischer Kontext; $\varrho_0 = x \mapsto \texttt{bot}$; $\varrho_i = x \mapsto \texttt{c}_{A,i}\ x_{i,1} \ldots x_{i,\alpha(\texttt{c}_{A,i})}$ und $L_i = \varrho_i(D[\texttt{case}_A\ x\ e_1 \ldots e_{|A|}] \in C, R), \varrho_i \circ \sigma$; dann lautet die Regel

$$\frac{D[\texttt{case}_A\ x\ e_1 \ldots e_{|A|}] \in C, R, \sigma}{\varrho_0(D[\texttt{case}_A\ x\ e_1 \ldots e_{|A|}] \in C, R), x \in \texttt{Bot}, \sigma \mid L_1 \mid \cdots \mid L_{|A|}}$$

Die Regel erzeugt keine Blätter, in denen x eine FWHNF hat, denn in diesem Fall ist der erzeugte Λ-Ausdruck nicht wohlgetypt und kann sofort von der Regel für Typfehler behandelt werden, die ein *no!* hinzufügt.

Äquivalente Kontexte Kontexte mit gleicher Konkretisierung sind untereinander austauschbar. Dies betrifft unter anderem die Ersetzung eines Namens durch seine Definition und syntaktische Transformationen wie das Hochschieben von Vereinigungen in Richtung der Wurzel des Syntaxbaumes.

Unabhängige Constraints Ein Constraint $t \in A$ kann getrennt von den restlichen Constraints des Blattes bearbeitet werden. Ist eine Lösung $S = T$ für $t \in A$ bekannt, so kann dieses Constraint ersetzt werden durch $(x_1, \ldots, x_n) \in S$; $(x_1, \ldots, x_n)$ sind die freien Variablen in t.

Falls $T = \langle\rangle$, so kann das Blatt mit *no!* markiert werden.

Es ist möglich, daß mehrere Regeln auf einen Knoten anwendbar sind. In diesem Fall muß eine Heuristik bereitgestellt werden, die über die anzuwendende Regel entscheidet.

Schleifenregeln

Durch die Erkennung von Schleifen gewinnt der Kalkül an Mächtigkeit. Es gibt vier Schleifenregeln, die potentiell unendliche Berechnungen abfangen können.

Hier wird ein Konzept der *Größe* von Λ-Termen benötigt. Dazu definieren wir für Terme aus Variablen, Konstruktoren und `bot`:

$$\begin{aligned} size(\texttt{bot}) &:= 1 \\ size(\texttt{c}\ t_1 \ldots t_n) &:= 1 + \sum_i size(t_i) \\ size(x) &:= 0, \text{ wenn } x \text{ Variable} \end{aligned}$$

Einige weitere Begriffe werden gebraucht: Für Constraints R und S sagen wir R *impliziert* S, wenn jede Lösung von R (d.h. jede Lösung eines Knotens (R, id)) auch eine Lösung von S ist. R und S heißen voneinander *unabhängig*, wenn die Mengen ihrer freien Λ-Variablen disjunkt sind. Ein Kontext D, der weder direkt noch indirekt über Kontextdefinitionen `where`-Ausdrücke enthält, heißt `Bot`-*abgeschlossen*, wenn aus $C[t] \in \eta(D)$ (dabei sei $C[\cdot]$ sei ein beliebiger syntaktischer Kontext) folgt, daß für alle $t' \in \gamma(\texttt{Bot})$ gilt: $C[t'] \in \eta(D)$.

$B[\cdot, \ldots, \cdot]$ und $C[\cdot, \ldots, \cdot]$ seien im folgenden n- bzw. r-stellige Mehrlochkontexte (siehe Definition 3.31, Seite 20).

Sei die Wurzel mit $B[x_1, \ldots, x_n] \in E, S, id$ markiert und $T := C[x_1, \ldots, x_r] \in D, S, id$ sei ein Knoten oberhalb der ersten `case`-Verzweigung. Mit diesem werden die Schleifen gebildet werden. S sei dabei eine Menge von CCVs.

Das Blatt, um das es geht, sei mit $C[t_1^1, \ldots, t_r^1] \in D, \ldots, C[t_1^m, \ldots, t_r^m] \in D, R, \sigma$ markiert. Auch R bestehe nur aus CCVs. $\mathcal{P}$ sei der Pfad von der Wurzel zum Blatt.

Die vier Schleifenregeln lauten wie folgt:

1. *loop-size*

   Die folgenden Bedingungen mögen gelten:

- R impliziert die Constraints $\{ x_i \mapsto t_i^j \mid i = 1, \ldots, r \}(S)$ für alle j, d.h. die CCVs des Blattes widersprechen nicht den CCVs des Knotens, mit dem die Schleife eventuell gebildet wird,
- die t_i^j bestehen nur aus Variablen, Konstruktoren und `bot`,
- für alle j ist $\sum_i size(\sigma x_i) > \sum_i size(t_i^j)$,
- für alle j gilt: Jede Variable $y \in \mathcal{FV}(t_1^j, \ldots, t_r^j)$ kommt in $\sigma x_1, \ldots, \sigma x_r$ mindestens so oft vor wie in is $t_1^j, \ldots, t_r^j$,
- $C[t_1^i, \ldots, t_r^i] \in D$ and $C[t_1^j, \ldots, t_r^j] \in D$ sind unabhängig für $i \neq j$.

Dann kann ein Blatt angehängt werden, das mit *loop!* markiert ist zusammen mit dem `where`-Ausdruck $(\sigma x_1, \ldots, \sigma x_r, \underbrace{\texttt{Top}, \ldots, \texttt{Top}}_{n-r})$ `where` $((t_1^1, \ldots, t_n^1), \ldots, (t_1^m, \ldots, t_n^m)) = \underbrace{(N, \ldots, N)}_{m}$ und dem CCV R, wobei $t_i^j = y_i^j$ ist mit neuen Variablen y_i^j, $r+1 \leq i \leq n$, und N ist der Name der Kontextdefinition für den Ausdruck im oberen Schleifenknoten T.

2. *loop-decomp*

   Für alle i, j sei $\sigma x_i = t_i^j$, und R impliziere $\sigma(S)$. Man unterscheidet nun drei Fälle:

   (a) Ist der Kontext D `Bot`-abgeschlossen und gibt es eine Konstruktordekomposition auf $\mathcal{P}$, so entfernen wir die Constraints $C[t_1^1, \ldots, t_r^1] \in D, \ldots, C[t_1^m, \ldots, t_r^m] \in D$, und behalten also im hinzugefügten Blatt nur noch R, σ.

   (b) Wenn $\texttt{Bot} \leq D$ ist und auf $\mathcal{P}$ keine Konstruktordekomposition, jedoch mindestens eine abstrakte Reduktion stattfindet, dann können wie oben die Constraints $C[t_1^1, \ldots, t_r^1] \in D, \ldots, C[t_1^m, \ldots, t_r^m] \in D$ entfernt werden, so daß R, σ übrigbleiben.

   (c) Wenn $\texttt{Bot} \not\leq D$ ist und auf $\mathcal{P}$ keine Konstruktordekomposition, jedoch mindestens eine abstrakte Reduktion stattfindet, dann markieren wir das Blatt mit *no!*.

5.2 Neue Regeln für die Behandlung von Funktionen

Wenn im Laufe der Analyse eine Variable, die für ein Argument höherer Ordnung steht, ihrerseits auf Argumente angewendet wird, entstehen Anfragen, die der eben beschriebene Kalkül nicht lösen kann. Daher werden nun zusätzliche Regeln eingeführt, die Abhilfe schaffen.

Welche unlösbaren Anfragen sind das genau? Der Λ-Term f sei eine freie Variable oder eine Anwendung einer freien Variablen auf ein oder mehrere Argumente. $f\ t \in C$ ist nun ein Constraint, das mit den bisherigen Regeln nicht oder nur unwesentlich bearbeitet werden kann. Die einzigen anwendbaren Regeln sind solche, die den Kontext C modifizieren oder das Constraint ganz streichen. $f\ t$ selbst kann aber als Λ-Ausdruck nicht reduziert oder zerlegt werden, da er eine Applikation einer Variablen ist, eine sogenannte *abstrakte Applikation*. Der Term kann nie aufgelöst werden.

Die im Kapitel über Kontexte eingeführten $\overset{\forall}{\rightarrow}$-Ausdrücke können Aussagen über Funktionen machen und eignen sich dazu, mit diesem Fall umzugehen:

5.2.1 Regel zur Anwendungsauflösung

$$\frac{f\ t \in C, R, \sigma}{t \in y, f \in y \overset{\forall}{\rightarrow} C, R, \sigma}, \ y \text{ ist eine in } C \text{ und } R \text{ nicht vorkommende Kontextvariable.} \quad \text{(DissolveAp)}$$

Die Constraints im entstehenden Blatt können so interpretiert werden: „t kann einen beliebigen Wert annehmen, aber f muß mindestens diesen Wert in die Konkretisierung von C abbilden."

Kontextvariablen

Die obige Regel führt eine *Kontextvariable* ein. Sie stellt die Verbindung zwischen f und t her, die jetzt in getrennten Constraints erscheinen. Da ein Knoten des Tableaus nun solche Variablen enthalten kann, müssen auch die Lösungen solcher Knoten neu definiert werden. Auch Tableauwurzeln mit Kontextvariablen sind denkbar, weswegen hier die ganze Definition 5.3 modifiziert wird:

Definition 5.6 (Lösung mit Kontextvariablen). *Eine Substitution θ ist eine* Lösung des Knotens (rs, σ) *mit* $rs = \{s_1 \in C_1, \ldots, s_n \in C_n\}$ *und mit in rs vorkommenden Kontextvariablen* $y_1, \ldots, y_m$ *genau dann, wenn*

$$(\theta s_1, \ldots, \theta s_n) \in \gamma((C_1, \ldots, C_n) \texttt{ where } y_1 = \texttt{Top}; \ldots; y_m = \texttt{Top}).$$

Eine solche Lösung ist Lösung der Wurzel (ts, ϱ), $ts = \{t_1 \in D_1, \ldots, t_k \in D_k\}$ *mit Kontextvariablen* $(z_1, \ldots, z_l)$ *und damit des Tableaus, wenn*

$$(\theta\sigma t_1, \ldots, \theta\sigma t_k) \in \gamma((D_1, \ldots, D_k) \texttt{ where } z_1 = \texttt{Top}; \ldots; z_l = \texttt{Top}).$$

Das `where` hat hierbei die Aufgabe, für jede Variable ihre verschiedenen Vorkommen in rs aufeinander abzugleichen, so daß gleiche Variablen immer auf den gleichen primitiven Kontext abgebildet werden.

Die Definition kann so aufgefaßt werden, daß sie auch für Knoten ohne Kontextvariablen gilt. Das `where` fällt dann weg, und die Formeln entsprechen denen in der alten Definition 5.3.

Die Korrektheit und eventuelle Vollständigkeit aller bisher in der Kontextanalyse definierten Regeln wird durch diese Änderung der Definition nicht angetastet. Denn für jede Substitution, die aus dem `where`-Fall der Definition von η hervorgeht und die Kontextvariablen substituiert, läßt sich die alte Definition von Lösungen anwenden.

Bemerkung. *Eine Kontextvariable kommt in einem Knoten des Tableaus entweder genau zweimal oder gar nicht vor, denn von allen Regeln dieses Abschnitts werden neue Kontextvariablen nur paarweise eingeführt und nur jeweils alle Vorkommen einer Variablen entfernt.*

Im Rest dieses Kapitels und auch im nächsten werden wir häufig über Kontexte sprechen, von denen nicht bekannt ist, ob sie ein `where`-Ausdruck sind oder nicht. Dafür wird nun eine Notation eingeführt:

Schreibweise. Syntaktische Objekte, deren Auftreten nicht sicher, aber auch nicht auszuschließen ist, sind in eckigen Klammern gesetzt, sofern diese nicht mit Haskell-Listenklammern zu verwechseln sind.

Z.B. schreiben wir in diesem Kapitel C [`where ...`], wenn nicht bekannt ist, ob C freie Variablen enthält und somit in Zusammenhang mit Definition 5.6 ein `where`-Ausdruck gefordert ist.

Wir betrachten nun die Korrektheit und Vollständigkeit von (DissolveAp).

Satz 5.7. *Die Regel* (DissolveAp) *ist korrekt.*

Beweis. Zu zeigen ist, daß jede Lösung des unteren Teils der Regel auch eine Lösung des oberen Teils ist.

Sei θ eine Lösung von $t \in y, f \in y \overset{\forall}{\to} C, R, \sigma$ mit $R = \{s_1 \in C_1, \ldots, s_r \in C_r\}$; also $\theta(t, f, s_1, \ldots, s_r) \in \gamma((y, y \overset{\forall}{\to} C, C_1, \ldots, C_r) \texttt{ where } y = \texttt{Top}; \ldots)$.

D.h. es ist $\theta(t, f, s_1, \ldots, s_r) \equiv_c \bigsqcup_i^c z_i$ mit $\forall i : z_i \in \eta((y, y \overset{\forall}{\to} C, C_1, \ldots, C_r) \texttt{ where } y = \texttt{Top}; \ldots)$. Wir folgern weiter[1]

$$\begin{aligned}
&\Longrightarrow \quad \forall i : \exists \varrho_i \in \Sigma_p : z_i \in \eta(\varrho_i((y, y \overset{\forall}{\to} C, C_1, \ldots, C_r))) \\
&\Longrightarrow \quad \forall i : \exists \varrho_i \in \Sigma_p : z_{i,1} \in \eta(\varrho_i(y)) \wedge z_{i,2} \in \eta(\varrho_i(y \overset{\forall}{\to} C)) \\
&\qquad\qquad\qquad\qquad \wedge (z_{i,3}, \ldots, z_{i,r+2}) \in \eta(\varrho_i(C_1, \ldots, C_r)) \\
&\Longrightarrow \quad \forall i : \exists \varrho_i \in \Sigma_p : z_{i,2}\ z_{i,1} \in \eta(\varrho_i(C)) \wedge (z_{i,3}, \ldots, z_{i,r+2}) \in \eta(\varrho_i(C_1, \ldots, C_r)) \\
&\Longrightarrow \quad \forall i : ((z_{i,2}\ z_{i,1}), z_{i,3}, \ldots, z_{i,r+2}) \in \eta((C, C_1, \ldots, C_r)\ [\texttt{where } \ldots])
\end{aligned}$$

[1] Hier und auf den folgenden Seiten bezeichne $z_{i,j}$ die j-te Komponente des Jec-Konstruktes z_i.

Nach Lemma 3.33 ist $\theta(f\ t) \equiv_c \bigsqcup_i^c (z_{i,2}\ z_{i,1})$, und damit haben wir
$\theta(f\ t, s_1, \ldots, s_r) \in \gamma((C, C_1, \ldots, C_r)\ [\texttt{where}\ \ldots])$ ◇

Die Regel ist i.a. nicht vollständig: In einer Lösung des Blattes, in der t FWHNF hat, muß f zwingenderweise *alle* Funktionen nach C abbilden, da der `where`-Ausdruck, über den die Lösung definiert ist, die Ausdrücke innerhalb von `Fun` nicht unterscheidet. Im Elternknoten des Blattes ist diese Einschränkung jedoch nicht vorhanden.

Vollständigkeit kann für manche Lösungen bewiesen werden, bei denen t in einem Kontext enthalten ist, der nur aus `Bot` und Konstruktoren aufgebaut ist:

Satz 5.8. *Die Regel* (DissolveAp) *ist vollständig für Lösungen θ mit $\theta t \in \gamma(T)$, wobei T ein primitiver Kontext ohne* `Fun` *ist, und eine der beiden folgenden Bedingungen erfüllt ist: die Implikation* (4.6) $\Longrightarrow$ (4.5) *gilt für f, oder es ist $\theta\sigma(f\ t) \in \eta(C)$.*

Beweis. Sei θ eine Lösung des Elternknotens, d.h. $\theta(f\ t, s_1, \ldots, s_r) \in \gamma((C, C_1, \ldots, C_r)\ [\texttt{where}\ \ldots])$. Es ist also $\theta(f\ t, s_1, \ldots, s_r) \equiv_c \bigsqcup_i^c z_i$ mit $z_i \in \eta((C, C_1, \ldots, C_r)\ [\texttt{where}\ \ldots])$. Das bedeutet: $\forall i : \exists^2 \tau_i \in \Sigma_p : z_i \in \eta(\tau_i((C, C_1, \ldots, C_r)))$.

Es ist $\theta(f\ t) \in \gamma(C\ [\texttt{where}\ \ldots])$. Nach Lemma 4.8 sind alle Elemente in $\eta(T) = \gamma(T)$ äquivalent bezüglich $\equiv_c$, so daß $\forall y \in \gamma(T) : \theta f\ y \in \gamma(C\ [\texttt{where}\ \ldots])$ (denn γ ist über $\equiv_c$-Äquivalenz definiert). Gilt die Implikation (4.6) $\Longrightarrow$ (4.5), folgt $\theta f \in \gamma(T \overset{\forall}{\to} C\ [\texttt{where}\ \ldots])$. Damit ist $\theta f \equiv_c \bigsqcup_i^c f_i$ mit $\forall i : f_i \in \eta(T \overset{\forall}{\to} C\ [\texttt{where}\ \ldots])$. Mit $\sigma_i := \tau_i \circ (y \mapsto T)$ ist also $\forall i : (\theta t, f_i) \in \eta(\sigma_i(y, y \overset{\forall}{\to} C))$ und daher $\theta(t, f, s_1, \ldots, s_r) \in \gamma((y, y \overset{\forall}{\to} C\ C_1, \ldots, C_r)\ \texttt{where}\ y = \texttt{Top}; \ldots)$.

Liegt $\theta(f\ t)$ in $\eta(C)$, kann der Beweis genauso geführt werden, aber ohne die nötigen Eigenschaften aus der Benutzung von γ herleiten zu müssen. ◇

Die obige Regel *erzeugt* Kontextvariablen im Tableau. Nun geht es um Regeln, die die Variablen verarbeiten oder entfernen können.

5.2.2 Regel für Konstruktor mit Kontextvariablen

Eine weitere neue Regel entsteht durch Anpassung der Konstruktordekompositionsregel für Kontextvariablen:

$$\frac{\mathtt{c}\ t_1 \ldots t_n \in y, R, \sigma}{\mathtt{c}\ t_1 \ldots t_n \in \mathtt{c}\ y_1, \ldots, y_n, (y \mapsto \mathtt{c}\ y_1 \ldots y_n)(R), \sigma}$$

$n = \alpha(\mathtt{c})$, y Kontextvariable, die y_i neue, nicht in R vorkommende Variablen.

(ConsWithDVar)

Die Bezeichung der Regel lautet „ConsWithDVar“, da Kontextvariablen als `DVar` implementiert werden (siehe dazu Kapitel 7).

Der folgende Satz beweist Korrektheit und diesmal sogar Vollständigkeit der Regel.

Satz 5.9. *Die Regel* (ConsWithDVar) *ist korrekt und vollständig.*

Beweis. Sei $R = \{s_1 \in C_1, \ldots, s_r \in C_r\}$ und $F[\cdot]$ ein syntaktischer Kontext, dessen Loch ausnahmsweise abweichend zu Definition 3.3 für *alle* Vorkommen von y in $(y, C_1, \ldots, C_r)$ stehe, so daß $F[a] = (y \mapsto a)(y, C_1, \ldots, C_r)$.

- *Korrektheit:* Sei θ eine Lösung von $\mathtt{c}\ t_1 \ldots t_n \in \mathtt{c}\ y_1 \ldots y_n, (y \mapsto \mathtt{c}\ y_1 \ldots y_n)(R), \sigma$. Es gilt also $\theta((\mathtt{c}\ t_1 \ldots t_n, s_1, \ldots, s_r)) \in \gamma(F[\mathtt{c}\ y_1 \ldots y_n]\ \texttt{where}\ y_1 = \texttt{Top}; \ldots; y_n = \texttt{Top}; \ldots)$.

  Um $\theta((\mathtt{c}\ t_1 \ldots t_n, s_1, \ldots, s_r)) \in \gamma(F[y]\ \texttt{where}\ y = \texttt{Top}; \ldots)$ zu zeigen, genügt es, das Enthaltensein der Konkretisierung des einen Kontextes in der des anderen zu zeigen; nach der Definition des club reicht es aus, das gleiche für die primitiven Terme zu machen:

[2] Auch wenn in $(C, C_1, \ldots, C_r)$ keine Variablen vorkommen und es daher kein `where` braucht, denn dann ist τ_i sogar beliebig.

$\eta(F[\mathtt{c}\ y_1 \ldots y_n]\ \mathtt{where}\ y_1 = \mathtt{Top}; \ldots ; y_n = \mathtt{Top}; \ldots) \overset{!}{\subseteq} \eta(F[y]\ \mathtt{where}\ y = \mathtt{Top}; \ldots)$.

Diese Aussage ist nun relativ offensichtlich: Das `where` im zweiten Ausdruck kann für y unter anderem Terme $\mathtt{c}\ y_1 \ldots y_n$ einsetzen.

- *Vollständigkeit:* θ sei eine Lösung von $\mathtt{c}\ t_1 \ldots t_n \in y, R, \sigma$; zu zeigen ist, daß es auch eine Lösung von $\mathtt{c}\ t_1 \ldots t_n \in \mathtt{c}\ y_1, \ldots, y_n, (y \mapsto \mathtt{c}\ y_1 \ldots y_n)(R), \sigma$ ist.

  Die Beziehung $\theta((\mathtt{c}\ t_1 \ldots t_n, s_1, \ldots, s_r)) \in \gamma(F[y]\ \mathtt{where}\ y = \mathtt{Top}; \ldots)$ bedeutet, daß $\theta((\mathtt{c}\ t_1 \ldots t_n, s_1, \ldots, s_r)) \equiv_c \bigsqcup_i^c z_i$, wobei $\forall i : z_i \in \eta(F[y]\ \mathtt{where}\ y = \mathtt{Top}; \ldots)$. Damit gilt $\forall i : \exists \tau_i \in \Sigma_p : z_i \in \eta(\tau_i(F[y]))$. Die erste Komponente von $F[y]$ ist y. Deswegen ist $z_{i,1} \in \eta(\tau_i(y))$ und, da auch die $z_{i,1}$ eine Kette mit *club* bilden, $\bigsqcup_i^c z_{i,1} \equiv_c \theta(\mathtt{c}\ t_1 \ldots t_n)$. Mit einem Argument wie in Lemma 4.9 folgt, daß der primitive Kontext, den τ_i für y einsetzt, ab einem bestimmten i die Form $\mathtt{c}\ \ldots$ haben muß; o.B.d.A. gelte dies für alle i: $\tau_i(y) = \mathtt{c}\ A_i^1 \ldots A_i^n$ mit primitiven Kontexten A_i^j.

  Weiter läßt sich folgern

  $$\begin{array}{lll} \ldots & \Longrightarrow & \forall i : \exists \varrho_i \in \Sigma_p : z_i \in \eta(\varrho_i(F[\mathtt{c}\ y_1 \ldots y_n])) \\ & & \quad (\text{nämlich } \varrho_i = \tau_i \circ (y_1 \mapsto A_i^1) \circ \cdots \circ (y_n \mapsto A_i^n)) \\ & \Longrightarrow & \forall i : z_i \in \eta(F[\mathtt{c}\ y_1 \ldots y_n]\ \mathtt{where}\ y_1 = \mathtt{Top}; \ldots ; y_n = \mathtt{Top}; \ldots) \end{array}$$

  und schließlich

  $\bigsqcup_i^c z_i \equiv_c \theta(\mathtt{c}\ t_1 \ldots t_n, s_1, \ldots, s_r) \in \gamma(F[\mathtt{c}\ y_1 \ldots y_n]\ \mathtt{where}\ y_1 = \mathtt{Top}; \ldots ; y_n = \mathtt{Top}; \ldots)$

◇

5.2.3 Regel für `bot` mit Kontextvariablen

Schließlich erklären wir eine Regel für `bot` in Zusammenhang mit einer Kontextvariablen:

$$\frac{\mathtt{bot} \in y, R, \sigma}{(y \mapsto \mathtt{Bot})(R), \sigma},\ y \text{ Kontextvariable} \qquad \text{(BotWithDVar)}$$

Auch hier werden Korrektheit und Vollständigkeit bewiesen.

Satz 5.10. *Die Regel* (BotWithDVar) *ist korrekt und vollständig.*

Beweis. Sei wieder $R = \{s_1 \in C_1, \ldots, s_r \in C_r\}$ und $F[\cdot]$ ein syntaktischer Kontext, dessen Argument ein letztes Mal noch für *alle* Vorkommen von y in $(y, C_1, \ldots, C_r)$ stehe, so daß $F[a] = (y \mapsto a)(y, C_1, \ldots, C_r)$. Analog sei $F^*[a] = (y \mapsto a)(C_1, \ldots, C_r)$.

- *Korrektheit:* Sei θ eine Lösung von $(y \mapsto \mathtt{Bot})(R), \sigma$. Somit gilt $\theta((s_1, \ldots, s_r)) \in \gamma(F^*[\mathtt{Bot}]\ [\mathtt{where}\ \ldots])$, d.h. $\theta((s_1, \ldots, s_r)) \equiv_c \bigsqcup_i^c z_i$ mit $\forall i : z_i \in \eta(F^*[\mathtt{Bot}]\ [\mathtt{where}\ \ldots])$.

  Es folgt

  $$\begin{array}{lll} \forall i : \exists \varrho_i \in \Sigma_p : z_i \in \eta(\varrho_i(F^*[\mathtt{Bot}])) & \Longrightarrow & \forall i : \exists \tau_i \in \Sigma_p : z_i \in \eta(\tau_i(F^*[y])) \\ & & \quad (\text{nämlich } \tau_i = \varrho_i \circ (y \mapsto \mathtt{Bot})) \\ & \Longrightarrow & \forall i : \exists \tau_i \in \Sigma_p : (\mathtt{bot}, z_{i,1}, \ldots, z_{i,r}) \in \eta(\tau_i(F[y])), \end{array}$$

  also $\bigsqcup_i^c (\mathtt{bot}, z_{i,1}, \ldots, z_{i,r}) \equiv_c \theta(\mathtt{bot}, s_1, \ldots, s_r) \in \gamma(F[y]\ \mathtt{where}\ y = \mathtt{Top}; \ldots)$.

- *Vollständigkeit:* θ sei eine Lösung von $\mathtt{bot} \in y, R, \sigma$, also $\theta((\mathtt{bot}, s_1, \ldots, s_r)) \in \gamma(F[y]\ \mathtt{where}\ y = \mathtt{Top}; \ldots)$. Wir haben wieder den club einer Kette: $\theta((\mathtt{bot}, s_1, \ldots, s_r)) \equiv_c \bigsqcup_i^c z_i$ mit $\forall i : z_i \in \eta(F[y]\ \mathtt{where}\ y = \mathtt{Top}; \ldots)$. Damit gilt $\forall i : \exists \tau_i \in \Sigma_p : z_i \in \eta(\tau_i(F[y]))$. Nun ist y die erste Komponente von $F[y]$. Dadurch gilt $z_{i,1} \in \eta(\tau_i(y))$ und, da auch die $z_{i,1}$ eine Kette mit club bilden, $z_{i,1} \leq_c \theta\mathtt{bot} = \mathtt{bot}$. Somit muß der primitive Kontext, den τ_i für y einsetzt, $\tau_i(y) = \mathtt{Bot}$ sein.

  Weiter folgt:

$$
\begin{aligned}
\ldots &\implies \forall i : \exists \tau_i \in \Sigma_p : (z_{i,2}, \ldots, z_{i,r}) \in \eta(\tau_i(F^*[y])) \\
&\implies \forall i : \exists \tau_i \in \Sigma_p : (z_{i,2}, \ldots, z_{i,r}) \in \eta(\tau_i(F^*[\texttt{Bot}])) \\
&\implies \forall i : (z_{i,2}, \ldots, z_{i,r}) \in \eta(F^*[\texttt{Bot}]\ [\texttt{where}\ \ldots])
\end{aligned}
$$

und schließlich

$$\bigsqcup_i^c (z_{i,2}, \ldots, z_{i,r}) \equiv_c \theta(s_1, \ldots, s_r) \in \gamma(F^*[\texttt{Bot}]\ [\texttt{where}\ \ldots])$$

$\diamond$

Kapitel 6

Analysen mit Funktionskontexten

Die Kontextanalyse ist im letzten Kapitel eingeführt worden. Dort wurden bereits zusätzliche Regeln erklärt, die Kontexte der Form $C_1 \xrightarrow{\forall} C_2$ benutzen, um mit bisher nicht verarbeitbaren Constraints umzugehen.

Jene Funktionskontexte soll man aber auch in Anfragen an die Kontextanalyse formulieren können. Die Behandlung von $f \in C_1 \xrightarrow{\forall} C_2$ oder auch $f \in C_1 \xrightarrow{\exists} C_2$ wird in diesem Kapitel beschrieben.

6.1 Analyse von $f \in C_1 \xrightarrow{\forall} C_2$

Für die Analyse von $f \in C_1 \xrightarrow{\forall} C_2$ kann man versuchen, dieses direkt als Anfrage an den Kalkül zu formulieren. Um mit den Funktionskontexten umgehen zu können, braucht man dann neue Regeln, die diese Kontexte ($C_1 \xrightarrow{\forall} C_2$) auflösen und Zweige erzeugen, in denen ihre Bestandteile (C_1 und C_2) direkt vorkommen. Dazu müßte man neue Variablen einführen, die für das Argument von f stehen, und neuartige Bedingungen aufstellen ($f\ y \in C_2$ *für alle* $y \in C_1$; das ergibt eine neue Form von Knoten). Was aber passiert bei der Anwendung von Kalkülregeln, die einen Knoten mit mehreren Nachfolgern versehen, wie die `case`-Regeln und die Vereinigungsdekomposition? Die Bedingung „*für alle* ... " muß nicht mehr an jedem einzelnen Knoten gelten, sondern sie bezieht sich auf alle Nachfolgerknoten zusammen. Ein einzelner Nachfolgerknoten kann nun für sich, ohne Information über seine Geschwister, nichts sinnvolles mehr mit dieser Bedingung anfangen; ein Unterbaum, der aus ihm resultiert, entsteht ohne Beachtung der dabei mitgeschleppten Bedingung.

Man kann auch ohne eine neue Form von Knoten auskommen, indem man, statt obige Anfrage direkt zu verarbeiten, den allgemeineren Fall $f\ y \in C_2$ betrachtet und aus dessen Ergebnis dann das gesuchte herleitet. Es werden im Ablauf des Kalküls eventuell mehr Informationen generiert als mit der obigen Methode; doch die Arbeit, die hier nach dem Lauf des Kalküls getan wird, muß auch dort in irgendeiner Weise über die „*für alle* ... "-Knoten laufen. Uns erscheint die zweite Methode unkomplizierter.

Die grobe Vorgehensweise wird also folgendermaßen aussehen:

1. Analysiere $f\ y \in C_2$. Der Ergebniskontext heiße $f\text{-}C_2$.
2. Berechne aus $f\text{-}C_2$ den gesuchten Kontext für die Variablen in f.

Das Prinzip im zweiten Schritt wird sein, die Stelle aus $f\text{-}C_2$, die dem Argument y entspricht (d.h. die Stelle am weitesten rechts), auf das Enthalten von C_1 zu überprüfen und für die entsprechenden Teile von $f\text{-}C_2$ die dazugehörigen anderen Stellen in das Resultat aufzunehmen. Das ist aber nur das Grobprinzip; mit dieser einfachen Formulierung ist es noch nicht getan. Zwei Spezialfälle sind allerdings leicht: Ist $\eta(C) = \emptyset$, so können wir $(\texttt{Top}, \ldots, \texttt{Top})$ ausgeben; und wenn C_1 nur aus `Bot` und Konstruktoren besteht, kann C_1 Bestandteil eines Kontextmusters sein und $(x_1, \ldots, x_n)$ `where` $(x_1, \ldots, x_n, C_1) = f\text{-}C_2$ ist die Lösung.

6.1.1 Beispiele und ein Verfahren für einfache Fälle

Als ein erstes Beispiel betrachten wir die Anfrage `append` $xs \in$ `Fin`$\overset{\forall}{\to}$`Inf`. Im ersten Schritt lösen wir `append` $xs\ ys \in$ `Inf`; das ergibt

$$\texttt{app-Inf} = \left\langle \begin{array}{l} (\texttt{Bot}, \texttt{Top}), \\ (\texttt{[]}, \texttt{Inf}), \\ (\texttt{Top:}zs, ys)\ \texttt{where}\ (zs, ys) = \texttt{app-Inf} \end{array} \right\rangle.$$

Die Frage ist nun, welcher Kontext C steht für die Ausdrücke, für die folgendes gilt: $x \in \gamma(C)$ genau dann, wenn für alle $y \in \gamma($`Fin`$)$: `append` $x\ y \in \gamma($`Inf`$)$.

In der „rechtesten" Stelle von `app-Inf` stehen `Top`, `Inf` bzw. die Variable ys mit `where`-Ausdruck.

- Da alle Elemente in der Konkretisierung von `Fin` von dem Kontext in der „rechtesten" Komponente der ersten Zeile, `Top`, repräsentiert werden, nehmen wir den Kontext `Bot`, der zusammen mit `Top` einen Jec bildet, in die Lösung auf.
- Für die nächste „rechte Seite", `Inf`, sieht es anders aus: `Fin` $\not\leq$ `Inf`. Daher nehmen wir das zu `Inf` gehörende `[]` nicht in die Lösung auf.
- Der dritte zu beachtende Kontext ist ys, eine Variable. Mit ihr alleine können wir nichts anfangen. Wo kommt ys aber her? Darin steckt eine Rekursion in die selbe „rechteste" Stelle des gleichen Kontextes `app-Inf`. Wir gehen hier intuitiv vor und bilden im Ergebnis ebenfalls eine einfache Rekursion in der Hoffnung, daß dadurch genau die gewünschten, fehlenden Terme erzeugt werden: `Top:`zs `where` $zs = C$ kommt in die Lösung.

Das Resultat lautet damit

$$C = \left\langle \begin{array}{l} \texttt{Bot}, \\ \texttt{Top:}zs\ \texttt{where}\ zs = C \end{array} \right\rangle.$$

Da C konventionell ist, kann der `where`-Teil vereinfacht werden, und es ergibt sich die äquivalente Definition

$$\begin{aligned} C &= \langle \texttt{Bot}, \texttt{Top:}C \rangle \\ &\equiv \texttt{Inf}. \end{aligned}$$

Die Anfrage `append` $xs \in$ `[]`$\overset{\forall}{\to}$`Inf` hätte mit dieser Methode die gleiche Lösung C.

Die Anfrage `append` $xs \in \langle$`Bot, Top:Bot`$\rangle\overset{\forall}{\to}$`Inf` würde zunächst ebenfalls den Kontextkalkül für `append` $xs\ ys \in$ `Inf` anwerfen mit dem Zwischenergebnis `app-Inf`. Dann aber wird $\langle$`Bot, Top:Bot`$\rangle \leq i$ für die rechtesten Komponenten i von `app-Inf` getestet. Im Unterschied zu `append` $xs \in$ `Fin`$\overset{\forall}{\to}$`Inf` bemerkt die Analyse hier, daß $\langle$`Bot, Top:Bot`$\rangle \leq$ `Inf` gilt für den zweiten Eintrag von `app-Inf`, und übernimmt die korrespondierende Komponente `[]` in das Ergebnis, das somit folgendes Aussehen hat:

$$C' = \left\langle \begin{array}{l} \texttt{Bot}, \\ \texttt{[]}, \\ \texttt{Top:}C' \end{array} \right\rangle,$$

eine Repräsentation der Menge aller mit `[]` oder `bot` endenden Listen.

Die erste Unstimmigkeit sehen wir im folgenden Fall: Nehmen wir an, daß die Definition (bzw. das Analyseergebnis) von `app-Inf` ein wenig anders formuliert ist, nämlich

$$\texttt{app-Inf'} = \left\langle \begin{array}{l} (\texttt{Bot}, \texttt{Top}), \\ (\texttt{[]}, \texttt{Bot}), \\ (\texttt{[]}, \texttt{Top:Inf}), \\ (\texttt{Top:}zs, ys)\ \texttt{where}\ (zs, ys) = \texttt{app-Inf'} \end{array} \right\rangle.$$

Diese Aufteilung von `Inf` in `Bot` und `Top:Inf` könnte bei einer anderen Heuristik des Kalküls oder bei einer komplizierteren Anfrage durchaus auftauchen. Unser naives Vorgehen würde dann — im Gegensatz zu vorher — in der Lösung zu `append` $xs \in \langle$`Bot,Top:Bot`$\rangle \overset{\forall}{\to}$`Inf` nicht die leere Liste `[]` aufführen, da weder $\langle$`Bot,Top:Bot`$\rangle \leq$ `Bot` noch $\langle$`Bot,Top:Bot`$\rangle \leq$ `Top:Inf` gilt. Offenbar ist bei der Darstellung eines größeren Kontextes (`Inf`) durch kleinere (`Bot` und `Top:Inf`) besondere Aufmerksamkeit geboten.

Genauer gesagt, muß es berücksichtigt werden, wenn Terme in der Konkretisierung der „linken" Komponenten von gleich mehreren der Jecs, die in der Kontextvereinigung f-C_2 stehen, enthalten sind. Dann können die korrespondierenden Stellen auf der rechten Seite möglicherweise insgesamt den Kontext C_1 umfassen, auch wenn sie es einzeln nicht tun.

Man muß also solche Einträge von f-C_2 beachten, deren rechte Stelle bezüglich der Konkretisierung nicht C_1 umfaßt. Eine Kombination dieser Einträge, die die gemeinsamen „linken" Komponenten der ursprünglichen Einträge besitzt und in deren rechten Seite aber die Vereinigung der rechten Seiten der ursprünglichen Einträge steht, kann nun erfolgversprechender mit dem ersten, naiven Ansatz angegangen werden. Diese Kombinationen sind die „zusätzlichen Einträge" in Schritt 2 des folgenden Verfahrens. Die Einträge (`[]`,`Bot`) und (`[]`,`Top:Inf`) aus dem obigen Beispiel würden z.B. kombiniert zu (`[]` $\cap$ `[]`,$\langle$`Bot,Top:Inf`$\rangle$). Die linke Stelle ist hier äquivalent zu `[]`, und in der rechten Stelle ist nun $\langle$`Bot,Top:Bot`$\rangle$ enthalten.

Im Detail sieht eine Konstruktion der gesuchten Lösung für den Fall, daß f-C_2 keine `where`-Einträge enthält, folgendermaßen aus:

Verfahren 6.1 (Analyse von $f \in C_1 \overset{\forall}{\to} C_2$ ohne `where`-Berücksichtigung).

1. f habe n Variablen.

   Falls $\eta(C_1) = \emptyset$, ist

   $$f\text{-}C_2\text{-}C_1 = (\underbrace{\texttt{Top},\ldots,\texttt{Top}}_{n\text{-mal}})$$

   die Lösung und es muß nichts mehr gemacht werden.

   Sonst analysiere $f\ y \in C_2$; y sei eine nicht in f vorkommende Variable. Das Ergebnis wird unter dem Namen f-C_2 in die Menge der Kontextdefinitionen aufgenommen. Es ist eine Vereinigung von Joint-evaluation-Kontexten (o.B.d.A., denn solche Formen entstehen durch die Kontextanalyse); es soll kein `where`-Kontext unter ihnen sein:

   $$f\text{-}C_2 = \left\langle \begin{array}{c} (x_{1,1},\ldots,x_{1,n+1}), \\ \vdots \\ (x_{l,1},\ldots,x_{l,n+1}) \end{array} \right\rangle$$

2. Falls sich C_1 nur aus `Bot` und Konstruktoren zusammensetzt, ist das Ergebnis

   $$f\text{-}C_2\text{-}C_1 = (x_1,\ldots,x_n)\ \texttt{where}\ (x_1,\ldots,x_n,C_1) = f\text{-}C_2$$

   und wir sind fertig.

   Andernfalls sei $I = \{\, i \,|\, C_1 \not\leq x_{i,n+1} \,\}$. Definiere nun $\widehat{f\text{-}C_2}$ als eine Kontextvereinigung mit den gleichen Einträgen wie f-C_2 sowie den zusätzlichen Einträgen

   $$(\cap_{j\in M} x_{j,1},\ldots,\cap_{j\in M} x_{j,n},\langle x_{j,n+1}\rangle_{j\in M})$$

   für alle $\emptyset \neq M \in \mathcal{P}(I)$.

   $\widehat{f\text{-}C_2}$ hat die Form

   $$\widehat{f\text{-}C_2} = \left\langle \begin{array}{c} (x'_{1,1},\ldots,x'_{1,n+1}), \\ \vdots \\ (x'_{l',1},\ldots,x'_{l',n+1}) \end{array} \right\rangle.$$

3. Das Ergebnis ist
$$f\text{-}C_2\text{-}C_1 = \langle\, (x'_{i,1}, \ldots, x'_{i,n}) \mid C_1 \leq x'_{i,n+1} \,\rangle.$$
(Die $\langle \ldots | \ldots \rangle$-Schreibweise ist wie bei Mengen zu verstehen.)

Die Vollständigkeit und Korrektheit des Verfahrens wird weiter unten behandelt nach einer Verallgemeinerung des Verfahrens.

Für die nächsten Abschnitte sei an die in Kapitel 5 eingeführte Schreibweise erinnert, die uns z.B. erlaubt, S_i [`where` $T_i = N_i$] zu schreiben, wenn zu manchen S_i ein `where` gehört, zu anderen nicht, und das i aber nicht konkret bekannt ist.

6.1.2 Verallgemeinerung des Verfahrens

`where`-Ausdrücke im Vereinigungskontext f-C_2 müssen, sofern C_1 nicht nur aus `Bot` und Konstruktoren besteht, gesondert behandelt werden. Die wesentliche Aufgabe, das Testen von $C_1 \leq rs$, wobei rs die jeweils „rechteste" Stelle eines Joint-evaluation-Kontextes darstellen soll, ist hier einerseits schwieriger als ohne `where` und führt andererseits nicht unbedingt zum gewünschten Ergebnis, da sich hinter einem `where`-Ausdruck ein komplizierteres Gebilde verstecken kann als nur ein simpler Joint-evaluation-Kontext, beispielsweise wieder eine Vereinigung.

- Der einfachste Fall ist hierbei, daß rs keine durch das `where`-Kontextmuster gebundenen Variablen enthält. In diesem Fall kann Verfahren 6.1 benutzt werden; die Vereinigung und Schnittmengenbildung im 2. Schritt muß allerdings den `where` $T = N$-Teil berücksichtigen. Bei Zusammentreffen zweier `where`-Kontexte entsteht ein neues `where` mit eventueller Umbenennung der alten Variablen.

- Enthält rs eine Variable, so kann es zunächst möglich sein, $C_1 \leq rs$ trotz der Variablen zu entscheiden: zum Beispiel, wenn diese Ungleichung nicht von den eingesetzten Werten der Variablen abhängt.

  - Bei der Antwort „Nein" können wir wie im Fall ohne `where` weitermachen: nämlich den Eintrag in das Schneiden/Vereinigen miteinbeziehen und ihn selbst nicht in die Lösung übernehmen.
  - Ist die Antwort „Ja", gibt es zunächst die einfache Möglichkeit, daß die Variablen in rs die einzigen Variablen sind, die im zugehörigen Jec vorkommen. Dann kann man wie im ersten Verfahren die anderen Komponenten des Jec für das Ergebnis verwenden; das `where` wird für diese Komponenten nicht gebraucht.

    Die andere Möglichkeit, bei der wie in Verfahren 6.1 vorgegangen werden kann, ist gegeben, wenn der Kontext N aus dem `where`-Teil ein konventioneller Kontext ohne `where` und ohne Vereinigungen ist. Das garantiert nämlich unter anderem, daß die „linken" und „rechten" Komponenten voneinander unabhängig sind. Die „linken" Komponenten werden dann in die Lösung übernommen. Das erledigt eine Projektion.

    Ein Beispiel zur zweiten Möglichkeit: Die rechte Seite von (`Top:`zs, ys) `where` $(zs, ys) =$ `app-Inf'` aus dem `append`-Beispiel enthält zwar `Fin`, aber `app-Inf'` enthält eine Vereinigung. Das Übernehmen einer passenden linken Seite, `Top:`zs `where` $zs = C$, in die Lösung wie im obigen Verfahren wäre im übrigen auch falsch, da diese auch `Top:` $\square$ enthält.
  - In jedem anderen Fall wird der betreffende `where`-Kontext zunächst wie im ersten Verfahren mit für die Schnitt-/Vereinigungsbildung berücksichtigt und dann für die Lösung ein im folgenden beschriebener Kontext erzeugt, der ebenfalls ein `where` enthält.

Der allgemeine `where`-Fall

Es ist zu klären, wie der Kontext aus dem letzten Unterpunkt aussieht.

Der Eintrag in f-C_2 habe die Form $(x_1, \ldots, x_{n+1})$ `where` $T = N$. Zunächst transformieren wir die bewußte Bedingung $C_1 \leq x_{n+1}$ in eine äquivalente Bedingung $D \leq P(a)$, wobei a eine durch das `where` gebundene Variable ist und $P(a)$ das Kontextmuster an der Jec-Komponente von T, in der sich a befindet. Dieser Kontext D wird dann anstatt C_1 für eine rekursive Analyse benutzt werden. C_1 und der ganze Kontext f-C_2 sollen konventionell sein; damit wird auch D konventionell, und das brauchen wir später. x_{n+1} soll nur die eine Variable a enthalten. Weiterhin soll a nur jeweils einmal in $(x_1, \ldots, x_{n+1})$ und in T vorkommen.

Wie läuft die Transformation ab? Natürlich sind Aussagen wie „$C_1 \leq x_{n+1}$“ wegen darin vorkommender ungebundener Variablen syntaktisch gar nicht erlaubt und damit formal unbrauchbar. Wir müssen also erst einmal formalisieren.

Mit der folgenden Kontextmenge wird in dieser Transformation gearbeitet:

Definition 6.1 ($\Lambda^{C,X}$). $\Lambda^{C,X} := \{\textit{konventionelle Kontexte } X \textit{ mit } \eta(X) \neq \emptyset\}$.

Das Verhältnis zwischen C_1 und D sei nun durch die Formel

$$\forall X \in \Lambda^{C,X} : \eta(C_1) \subseteq \eta(x_{n+1} \texttt{ where } a = X) \iff \eta(D) \subseteq \eta(P(X)) \tag{6.1}$$

definiert. Um D nun aus den anderen Daten herleiten zu können, führen wir schrittweise Äquivalenzumformungen durch zwischen Aussagen, von denen jede die Form

$$\forall X \in \Lambda^{C,X} : \eta(C) \subseteq \eta(t(a) \texttt{ where } a = X) \iff \eta(D') \subseteq \eta(X) \tag{6.2}$$

hat. Ist erst einmal ein D' gefunden, das (6.2) erfüllt, ist damit für Gleichung (6.1) auch D mit $D \equiv P(D')$ gegeben.

Am Anfang steht die Aussageform (6.2) mit $C = C_1$ und $t(a) = x_{n+1}$. Für den Transformationsschritt unterscheiden wir nun nach der äußeren Form von t:

- $t(a) = a$: Die Beziehung $\forall X \in \Lambda^{C,X} : \eta(C) \subseteq \eta(a \texttt{ where } a = X) \iff \eta(D') \subseteq \eta(X)$ formen wir nicht weiter um; statt dessen können wir $D' := C$ setzen (keine Kontextdefinition; syntaktische Gleichheit ist gemeint!). Da X konventionell ist, ist nach Korollar 4.42 $\eta(X) = \eta(a \texttt{ where } a = X)$ und (6.2) gilt.

In den restlichen Fällen können wir in der Beziehung (6.2) den linken Teil $\eta(C) \subseteq \eta(t(a) \texttt{ where } a = X)$ durch eine ihm logisch äquivalente Ungleichung ersetzen. Wir zeigen nur die Äquivalenz des ersetzten Teilausdrucks mit dem Substitut; daraus folgt die Äquivalenz der gesamten Beziehung.

- $t(a) = \texttt{c}\ t_1 \ldots t_n$: Es ist

$$\eta(C) \subseteq \eta(\texttt{c}\ t_1 \ldots t_n \texttt{ where } a = X) \iff$$
$$\begin{cases} \forall i : \eta(p_{i,\texttt{c}}(C)) \subseteq \eta(t_i\ [\texttt{where } a = X]) & \text{wenn die } p_{i,\texttt{c}}(C) \text{ definiert sind} \\ \textit{Falsch} & \text{sonst.} \end{cases} \tag{6.3}$$

  ($p_{i,\texttt{c}}$ ist die kleinliche Projektion auf Kontexten bezüglich des Konstruktors `c` auf dessen i-ten Datenteil.)

  Das ist leicht zu sehen, falls die $p_{i,\texttt{c}}(C)$ undefiniert sind. Der andere Fall wird folgendermaßen bewiesen:

  Zunächst kann man leicht feststellen, daß, weil a nur in einem der t_i vorkommt, $\eta(\texttt{c}\ t_1 \ldots t_n \texttt{ where } a = X) = \eta(\texttt{c}\ t_1 \ldots t_{k-1}\ (t_k \texttt{ where } a = X)\ t_{k+1} \ldots t_n)$ ist, wobei t_k diejenige Komponente mit der Variable ist. Damit geht es wie folgt weiter:

  - „$\Longrightarrow$“: Wie Lemma 4.54 bezeugt, gilt für konventionelle Kontexte C und C', wenn $p_{i,\texttt{c}}$ auf ihnen definiert ist: $\eta(C) \subseteq \eta(C') \implies \forall i : \eta(p_{i,\texttt{c}}(C)) \subseteq \eta(p_{i,\texttt{c}}(C'))$. Dies kann man direkt auf unseren Fall anwenden.

– „$\Longleftarrow$“: Aus $\forall i : \eta(p_{i,\mathtt{c}}(C)) \subseteq \eta(t_i\ [\mathtt{where}\ a = X])$ folgt $\eta(\mathtt{c}\ p_{1,\mathtt{c}}(C) \ldots p_{n,\mathtt{c}}(C)) \subseteq \eta(\mathtt{c}\ t_1 \ldots t_{k-1}\ (t_k\ \mathtt{where}\ a = X)\ t_{k+1} \ldots t_n)$. Weiterhin gilt, da die $p_{i,\mathtt{c}}$ existieren, $\eta(C) \subseteq \eta(\mathtt{c}\ p_{1,\mathtt{c}}(C) \ldots p_{n,\mathtt{c}}(C))$. Diese beiden Aussagen kombinieren sich zum gewünschten Ergebnis.

Hinter dem Äquivalenzzeichen stehen mehrere ver-undete Aussagen; nur eine jedoch enthält die Variable a und hat die übliche Form; alle anderen enthalten weder a noch X, können daher weggelassen werden, sofern sie wahr sind. Ist eine falsch, gilt $\eta(C) \not\subseteq \eta(\mathtt{c}\ t_1 \ldots t_n\ \mathtt{where}\ a = X)$, und zwar sogar für jedes X. (Das gleiche trifft zu, wenn oben der zweite Fall „*Falsch*“ eintritt, wenn $p_{i,\mathtt{c}}(C)$ nicht definiert ist.) Das heißt, daß ungeachtet dessen, welchen Wert X die Variable a durch das `where` bekommt, immer $\eta(C) \not\subseteq \eta(t(a)\ \mathtt{where}\ a = X)$ gilt. Somit muß der Eintrag $(x_1, \ldots, x_{n+1})\ \mathtt{where}\ T = N$ gar nicht hier behandelt werden, sondern man kann ihn, wie weiter oben beschrieben, so wie im Verfahren 6.1 analysieren.

- $t(a) = \langle u, v(a)\rangle$: Die folgenden Äquivalenzumformungen gelten:

$$\begin{aligned}
\eta(C) \subseteq \eta(\langle u, v(a)\rangle\ \mathtt{where}\ a = X) &\Longleftrightarrow \eta(C) \subseteq \bigcup_{\sigma:\Lambda_p(\sigma a)\subseteq\eta(X)} \eta(\langle u, v(\sigma a)\rangle) \\
&\Longleftrightarrow \eta(C) \subseteq \bigcup_{\sigma:\Lambda_p(\sigma a)\subseteq\eta(X)} \eta(u) \cup \bigcup_{\sigma:\Lambda_p(\sigma a)\subseteq\eta(X)} \eta(v(\sigma a)) \\
&\Longleftrightarrow \eta(C \cap u^c) \subseteq \eta(v(a)\ \mathtt{where}\ a = X)
\end{aligned}$$

Die letzte Äquivalenz gilt nur dann, wenn $\eta(u) = \bigcup_{\sigma:\Lambda_p(\sigma a)\subseteq\eta(X)} \eta(u)$. Dies ist aber mit Korollar 4.42 gegeben durch die Quantifizierung von X mit $\eta(X) \neq \emptyset$ und X konventionell.

- $t(a) = u \cap v(a)$: Es ist

$$\eta(C) \subseteq \eta(u \cap v(a)\ \mathtt{where}\ a = X) \iff \begin{cases} \eta(C) \subseteq \eta(v(a)\ \mathtt{where}\ a = X) & \text{wenn } \eta(C) \subseteq \eta(u) \\ \textit{Falsch} & \text{wenn } \eta(C) \not\subseteq \eta(u) \end{cases}$$

Im zweiten Fall folgt aus der Gesamtformel (6.2) offenbar die Beziehung $\forall X \in \Lambda^{C,X} : \eta(D') \not\subseteq \eta(X)$. Da man auch $X = \mathtt{Top}$ einsetzen kann, gibt es für D' keine Lösung. Das heißt, daß wir an dieser Stelle nicht weitermachen können. Jedoch spiegelt dieser Fall nur den gleichen Sachverhalt wider, der oben bei $t(a) = \mathtt{c}\ t_1 \ldots t_n$ schon beschrieben wurde: Dieser Eintrag $(x_1, \ldots, x_{n+1})\ \mathtt{where}\ T = N$ gehört nicht hierher, sondern wird in einem anderen Fall des Verfahrens behandelt.

Der zu erzeugende Eintrag für das Analyseergebnis ist nun

$$(x_1, \ldots, x_n)\ \mathtt{where}\ T^* = N^*,$$

wobei T^* aus T entsteht durch Entfernen der Komponente, in der a vorkommt. N^* ist der Name des Ergebnisses der Analyse von N, wie sie hier auf $f\text{-}C_2$ durchgeführt wird, mit dem Unterschied, daß es bei der Analyse von N nicht mehr um die „rechteste“ Stelle der Kontexte geht, sondern um die Position, die in T durch die Variable a, die in x_{n+1} vorkommt, gekennzeichnet ist. Der zweite Unterschied ist, daß bei dieser rekursiven Analyse der Part von C_1 nun von dem oben beschriebenen D übernommen wird.

Da die zu untersuchende Stelle (hier x_{n+1}) höchstens eine Variable enthalten soll, die in T auch nur in einer Stelle vorkommt, kann bei der rekursiven Analyse von N als Parameter die Stelle angegeben werden, in der die Variable aus x_{n+1} in T vorkommt und die anstatt der „rechtesten“ Stelle verwendet werden soll. Diese Verallgemeinerung ist in der folgenden Beschreibung eines Analyseverfahrens enthalten.

Wir beleuchten vorher noch, worauf sich die oben geforderten Beschränkungen gründen:

- Wenn die Variable a mehrfach in $(x_1, \ldots, x_{n+1})$ vorkommt, könnte z.B. in dem Fall, daß sich das zusätzliche Variablenvorkommen an einer anderen Stelle des Jecs etwa in $\langle \texttt{Top}, a \rangle$ befindet, die Analyse recht einfach geführt werden. Aber wenn a alleine an einer Stelle des Jecs steht, zusätzlich zu dem Vorkommen in x_{n+1}, gibt es überhaupt nur Lösungen, wenn C ein primitiver Kontext ist. Denn in diesem Fall sind die beiden Stellen, in denen a steht, durch die `where`-Bindung nicht mehr unabhängig voneinander, bzw. nur noch bei Einsetzung eines primitiven Kontextes für a, weil diese durch das `where` nicht getrennt werden. Eine allgemeine Lösung für diese Fälle scheint schwierig zu formulieren.
- Kommen in x_{n+1} mehr als eine Variable vor, so könnte eine Bildung der Vereinigungs- und Schnittkontexte sehr schwierig werden, da dann eventuell mehrere Stellen in T und damit in der rekursiv aufgerufenen Analyse betrachtet werden müssen. Das gleiche trifft zu für mehrfaches Vorkommen von a in T.
- Wenn in der Komponente von T, in der a steht, noch eine andere Variable vorkommt, dann müßte weiter in die Struktur von N hineingegriffen werden als nur in die einzelnen Jec-Komponenten. Dies würde höchstwahrscheinlich zur Endlosrekursion führen.

Diese Einschränkungen können möglicherweise alle aufgehoben werden, indem die Analyse noch weiter verallgemeinert wird. Ein solches Verfahren wäre aber mit Sicherheit deutlich komplexer als das hier beschriebene, weswegen wir im Rahmen dieser Arbeit die obigen Beschränkungen vorziehen, zumal auch damit sinnvoll gearbeitet werden kann.

Einschränkung der Art, daß gewisse Kontexte konventionell sein sollen, werden „nur“ für den Korrektheitsbeweis gebraucht. Dann sind nämlich die Sätze aus Abschnitt 4.6 anwendbar, deren Aussage man sonst extra fordern müßte.

Das Verfahren sieht nun allgemein so aus:

Verfahren 6.2 (Analyse von $f \in C_1 \overset{\forall}{\to} C_2$ mit `where`-Berücksichtigung).

1. f habe n Variablen.

   Falls $\eta(C_1) = \emptyset$, ist

   $$f\text{-}C_2\text{-}C_1 = (\underbrace{\texttt{Top}, \ldots, \texttt{Top}}_{n\text{-mal}})$$

   die Lösung, und wir sind fertig.

   Sonst analysiere $f\ y \in C_2$. Das Ergebnis wird unter dem Namen $f\text{-}C_2$ in die Menge der Kontextdefinitionen aufgenommen.

   Rufe den nächsten Schritt auf mit den Parametern C_1, $f\text{-}C_2$ und $n+1$ (die auf $C_1 \leq ()$ zu untersuchende Stelle). Das zurückgegebene Endergebnis bekommt den Namen $f\text{-}C_2\text{-}C_1$.

2. Parameter für diesen Schritt: Kontexte C, G, sowie p (die auf $C \leq ()$ zu untersuchende Stelle).

   G ist eine Vereinigung von Joint-evaluation-Kontexten (reine Jecs und solche innerhalb `where`-Kontexten):

   $$G = \left\langle \begin{array}{c} (x_{1,1}, \ldots, x_{1,m})\ [\texttt{where}\ T_1 = N_1], \\ \vdots \\ (x_{l,1}, \ldots, x_{l,m})\ [\texttt{where}\ T_l = N_l] \end{array} \right\rangle$$

   Falls sich C nur aus `Bot` und Konstruktoren zusammensetzt, ist das Ergebnis

   $$G' = \underbrace{(y_1, \ldots, y_m)}_{\text{ohne } y_p}\ \texttt{where}\ \underbrace{(y_1, \ldots, y_m)}_{C \text{ statt } y_p} = G$$

   und wir sind fertig.

Andernfalls benötigen wir zum weiteren Vorgehen einige Voraussetzungen:

Voraussetzung für jeden Eintrag in G, der von einem der Fälle von (6.4)–(6.10) behandelt wird, in dem $C \leq a_p$ [`where...`] ist:

- Der Eintrag ist konventionell.

Voraussetzungen für jeden `where`-Eintrag $(x_{w,1}, \ldots, x_{w,m})$ `where` $T_w = N_w$, der unter Fall (6.9) oder (6.10) (s.u.) fällt:

(a) In $x_{w,p}$ taucht genau eine freie Variable a auf.

(b) a kommt in keiner anderen Komponente $x_{w,i}$ vor.

(c) a taucht nur einmal im Muster T_w auf.

(d) Die Jec-Komponente von T_w, in der a steht, enthält keine andere Variable.

Voraussetzung, wenn ein `where`-Eintrag $(x_{w,1}, \ldots, x_{w,m})$ `where` $T_w = N_w$ unter Fall (6.10) (s.u.) fällt:

- C und G sind konventionell.

Einträge in G, die obige Bedingungen nicht erfüllen, werden ignoriert. G enthalte also ab hier nur noch solche Kontexte, auf die die Voraussetzungen zutreffen.

Sei

$$I = \{\, i \mid C \not\leq x_{i,p}\ [\texttt{where}\ T_i = N_i] \,\}.$$

Definiere nun $\hat{G}$ als eine Kontextvereinigung mit den gleichen Einträgen wie G sowie den zusätzlichen Einträgen

$$(\bigcap_{j \in M} x_{j,1}, \ldots, \bigcap_{j \in M} x_{j,p-1}, \langle x_{j,p} \rangle_{j \in M}, \bigcap_{j \in M} x_{j,p+1}, \ldots, \bigcap_{j \in M} x_{j,m})$$

für alle $\emptyset \neq M \in \mathcal{P}(I)$.

Diese Angabe der zusätzlichen Einträge ist nicht ganz richtig: Beim Kombinieren von Einträgen mit `where` müssen diese `where` $T_j = N_j$ übernommen werden und bei Namenskonflikten Variablen umbenannt werden.

$\hat{G}$ hat die Form

$$\hat{G} = \left\langle \begin{array}{c} (x'_{1,1}, \ldots, x'_{1,m})\ [\texttt{where}\ T'_1 = N'_1], \\ \vdots \\ (x'_{l',1}, \ldots, x'_{l',m})\ [\texttt{where}\ T'_{l'} = N'_{l'}] \end{array} \right\rangle.$$

Definiere nun die Funktion F mit

$$
\begin{aligned}
F((a_1,\ldots,a_m)) &= \langle\rangle, \text{ falls } C \not\leq a_p && (6.4)\\
F((a_1,\ldots,a_m)) &= \Pi_{\hat{p}}((a_1,\ldots,a_m)), \text{ falls } C \leq a_p && (6.5)\\
F((a_1,\ldots,a_m) \texttt{ where } T = N) &= F((a_1,\ldots,a_m)) \texttt{ where } T = N, && \\
&\quad \text{falls } a_p \text{ keine Variablen enthält und } C \leq a_p && (6.6)\\
F((a_1,\ldots,a_m) \texttt{ where } T = N) &= \langle\rangle, \text{ falls } C \not\leq a_p \texttt{ where } T = N && (6.7)\\
F((a_1,\ldots,a_m) \texttt{ where } T = N) &= \Pi_{\hat{p}}((a_1,\ldots,a_m)), \text{ falls } C \leq a_p \texttt{ where } T = N, && \\
&\quad \text{und in } \Pi_{\hat{p}}((a_1,\ldots,a_m)) \text{ kommen keine} && \\
&\quad \text{Variablen vor} && (6.8)\\
F((a_1,\ldots,a_m) \texttt{ where } T = N) &= \Pi_{\hat{p}}((a_1,\ldots,a_m)) \texttt{ where } T^* = N^{**}, \text{ falls } a_p && \\
&\quad \text{eine Variable enthält, } C \leq a_p \texttt{ where } T = N && \\
&\quad \text{und } N \text{ ist konventionell ohne } \texttt{where} \text{ und} && \\
&\quad \text{Vereinigungen} && (6.9)\\
F((a_1,\ldots,a_m) \texttt{ where } T = N) &= \Pi_{\hat{p}}((a_1,\ldots,a_m)) \texttt{ where } T^* = N^* \text{ sonst} && (6.10)
\end{aligned}
$$

Zur Definition von T^*, N^* und N^{**}: Sei x die Variable, die in a_p vorkommt, und p_x sei die Nummer der Stelle von T, in der x steht.

Dann ist $T^* := \Pi_{\widehat{p_x}}(T)$.

Es sei $P(x) = \Pi_{p_x}(T)$.

D sei der Kontext, für den das Analogon zu Gleichung (6.1) gilt: $\forall X \in \Lambda^{C,X} : \eta(C) \subseteq \eta(a_p(x) \texttt{ where } x = X) \iff \eta(D) \subseteq \eta(P(X))$,

N^* ist dann das Ergebnis der Analyse von N (d.h. dieser Schritt des Verfahrens) bezüglich der Parameter D, N und p_x.

N^{**} ist gleich $\pi_{\widehat{p_x}}(N)$.

Das Ergebnis der Analyse ist

$$G' = \langle\, F(a) \,|\, a \text{ ist Eintrag in } \hat{G} \,\rangle.$$

Es stellt sich die Frage, ob das Verfahren korrekt bzw. vollständig ist. Damit beschäftigen sich die nächsten beiden Abschnitte. Wenn diese Begriffe im folgenden verwendet werden, ist damit immer die *partielle* Korrektheit und Vollständigkeit gemeint: also für den Fall, daß das Verfahren terminiert bzw. wohldefiniert ist. Durch die Rekursion im Fall (6.10) kann es nämlich passieren, daß ein Analyseergebnis sich auf eine andere Analyse bezieht, welche ihrerseits eine weitere Analyse referenziert usf. In diesem Fall ist keine Wohldefiniertheit gegeben. Es ist Aufgabe der Implementierung, solche Endlosrekursionen rechtzeitig abzubrechen und dann beispielsweise ein leeres Teilergebnis $\langle\rangle$ einzusetzen.

Wir gehen im folgenden davon aus, daß in der Analyse gebildete Kontexte G die in Schritt 2 aufgeführten Voraussetzungen erfüllen. Diese Annahme beeinflußt nicht den Korrektheitsbeweis; jedoch ist andernfalls natürlich keine Vollständigkeit zu erwarten.

Die Prämisse für die Korrektheit bzw. Vollständigkeit des Verfahrens ist die entsprechende Eigenschaft des benutzten Zwischenergebnisses $f\text{-}C_2$.

6.1.3 Korrektheit

Korrektheit des Verfahrens heißt:

$$\gamma(f\text{-}C_2\text{-}C_1) \subseteq \{\, (x_1,\ldots,x_n) \,|\, f(x_1,\ldots,x_n) \in \gamma(C_1 \overset{\forall}{\to} C_2) \,\} \qquad (6.11)$$

sofern $f\text{-}C_2$ korrekt ist, d.h.

$$\gamma(f\text{-}C_2) \subseteq \{\, (x_1,\ldots,x_{n+1}) \,|\, f(x_1,\ldots,x_n)\ x_{n+1} \in \gamma(C_2) \,\} \qquad (6.12)$$

Wir werden nur einen schwächeren Satz beweisen, der aber vermutlich sehr nahe an die echte Korrektheit herankommt. Unten werden hinreichende Bedingungen angegeben, unter denen die echte Korrektheit gegeben ist.

Satz 6.2. *Es gelte* (6.12). *Dann ist*

$$\gamma(f\text{-}C_2\text{-}C_1) \subseteq \{\,(x_1,\ldots,x_n)\,|\,\forall x_{n+1} \in \gamma(C_1) : f(x_1,\ldots,x_n)\ x_{n+1} \in \gamma(C_2)\,\}.$$

Wann trifft hier die echte Korrektheit (6.11) zu?

1. Für $\eta(C_1) = \emptyset$ trifft (6.11) ohne weitere Voraussetzung zu.
2. Falls gilt $(\forall x_{n+1} \in \gamma(C_1) : f(x_1,\ldots,x_n)\ x_{n+1} \in \gamma(C_2)) \implies f(x_1,\ldots,x_n) \in \gamma(C_1 \overset{\forall}{\to} C_2)$ (das ist die Implikation (4.6) $\implies$ (4.5)), ist die echte Korrektheit gegeben.
3. Falls gilt $\forall x_{n+1} \in \eta(C_1) : f(x_1,\ldots,x_n)\ x_{n+1} \in \eta(C_2)$, so ist nach Lemma 4.37 ebenfalls echte Korrektheit gegeben. Dies betrifft also Funktionen f, die sozusagen nicht von selbst unendliche Strukturen erzeugen.

Die Implikation aus Punkt 2 ist möglicherweise immer gegeben: Einerseits wird sie natürlich von Funktionen erfüllt, die der Bedingung aus Punkt 3 genügen. Andererseits gilt sie auch für Funktionen wie `repeat`, das ja von sich aus unendliche Strukturen erzeugt und für die Punkt 3 nicht anwendbar ist: Denn es gilt z.B.

$$\forall x \in \gamma(C_1) : \texttt{repeat}\ x \in \gamma(\texttt{Inf})$$

für jedes C_1 wegen Lemma 4.25. `repeat` läßt sich aber auch als club der Funktionen f_i mit

$$f_i\ x\ =\ (x\texttt{:})^i\ \texttt{bot}$$

schreiben (denn siehe das Beispiel nach Lemma 3.32), und für diese Funktionen gilt offenbar

$$\forall x \in \eta(C_1) : f_i\ x \in \eta(\texttt{Inf}),$$

also $f_i \in \eta(C_1 \overset{\forall}{\to} \texttt{Inf})$. Insgesamt folgt, daß

$$\texttt{repeat} \in \gamma(C_1 \overset{\forall}{\to} \texttt{Inf}),$$

also genau der gewünschte Schluß der Implikation!

Man kann vermuten, daß die Folgerung (4.6) $\implies$ (4.5) so für alle Funktionen und Kontexte beweisbar ist, indem also eine Funktion, die unendliche Strukturen erzeugt, als club einer Kette dargestellt wird, deren Glieder nur Endliches hervorbringen. Dann wäre das obige Analyseverfahren sicher korrekt. Dies bleibt aber im Rahmen dieser Arbeit nur eine Vermutung, für die jedoch immerhin die obigen Anhaltspunkte sprechen.

Der Beweis des Satzes wird weiter unten nach dem nächsten Lemma und Korollar geführt.

Zuerst wird dafür die Korrektheit des zweiten Schrittes des Verfahrens für primitive Terme allgemein formuliert und bewiesen:

Lemma 6.3. *Sei G' der Name des Analyseergebnisses von Schritt 2 des Verfahrens bezüglich G, C, p.*

$$\eta(G') \subseteq \{\,(\underbrace{x_1,\ldots,x_m}_{\text{ohne } x_p}) \in \pi_{\hat{p}}(\eta(G))\,|\,\forall x_p \in \eta(C) : (x_1,\ldots,x_m) \in \eta(G)\,\}$$

Beweis. Dies ist der einzige der Sätze und Hilfssätze zur Korrektheit, in deren Aussage eine Beschränkung der Elemente auf $\pi_{\hat{p}}(\eta(G))$ vorkommt. Sie kommt nur zur Geltung, wenn $\eta(C) = \emptyset$ ist und ist daher hier durchaus angebracht: Der Fall $\eta(C) = \emptyset$ wird zwar anfangs von Schritt 1 des Verfahrens abgefangen, kann aber durch die Hintertür über eine `where`-Rekursion in Fall (6.10)

wieder in den hier behandelten zweiten Schritt gelangen, der für G' nicht $(\texttt{Top}, \ldots, \texttt{Top})$ setzt, sondern eine Projektion von G.

Zum Beweis für den Fall, daß C sich aus Konstruktoren und `Bot` zusammensetzt, sei ein $\sigma \in \Sigma_p$ mit $\Lambda_p(\sigma(\underbrace{y_1, \ldots, y_m}_{C \text{ statt } y_p})) \subseteq \eta(G)$ gegeben. Das impliziert $\forall(\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p}) \in \eta(\sigma(\underbrace{y_1, \ldots, y_m}_{\text{ohne } y_p}))$: $\forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \eta(G)$. Umformuliert heißt das gerade: $\eta(G') \subseteq \{ (\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p}) \mid \forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \eta(G) \}$. Die Quantifizierung auf Elemente aus $\pi_{\hat{p}}(\eta(G))$ ergibt sich daraus automatisch, da $\eta(C)$ hier nicht leer ist.

Nun zu den anderen Fällen. Zunächst merken wir an, daß weiter unten im schwierigsten Teil des Beweises, nämlich für den Fall der allgemeinen Rekursion bei `where`-Kontexten, Fall (6.10), die Ungleichung $\eta(C) \subseteq \eta(a_p(x)\ \texttt{where}\ x = D')$ benutzt wird, falls $\eta(D) \neq \emptyset$. (Dabei ist $D = P(D')$.) Dies ist der eigentliche Anknüpfungspunkt, an dem die Induktionsvoraussetzung im Korrektheitsbeweis des Rekursionsfalles aufgehängt werden kann, um den Beweis zu schließen. Die Beziehung kann hergeleitet werden, da D' in der Beschreibung des Verfahrens so konstruiert wurde, daß $\forall X \in \Lambda^{C,X} : \eta(C) \subseteq \eta(a_p(x)\ \texttt{where}\ x = X) \iff \eta(D') \subseteq \eta(X)$. Unter der Voraussetzung $\eta(D) \neq \emptyset$ (das trifft genau dann zu, wenn $\eta(D') \neq \emptyset$) und weil D ein konventioneller Kontext ist (denn C und der gesamte Kontext G, also auch N, sind als konventionell vorausgesetzt), kann es für X eingesetzt werden, woraus die zu zeigende Bedingung sofort folgt.

Wir kommen zum eigentlichen Beweis:

1. $\eta(\hat{G}) = \eta(G)$:

   „$\supseteq$“ geht aus der Definition von $\hat{G}$ hervor. Für „$\subseteq$“ betrachten wir einen Eintrag e der Vereinigung aus $\hat{G}$, der in G nicht vorkommt. Es ist (o.B.d.A. sei hier $p = m$)

   $$\begin{aligned} \eta(e) &= \eta((\textstyle\bigcap_{j\in M} x_{j,1}, \ldots, \bigcap_{j\in M} x_{j,m-1}, \langle x_{j,m}\rangle_{j\in M})\ [\texttt{where}\ \ldots]) \\ &= \eta(\langle(\textstyle\bigcap_{j\in M} x_{j,1}, \ldots, \bigcap_{j\in M} x_{j,m-1}, x_{j',m})\rangle_{j'\in M}\ [\texttt{where}\ \ldots]) && \text{(Vereinigung nach oben geschoben)} \\ &= [\textstyle\bigcup_\sigma] \bigcup_{j'\in M} \eta((\bigcap_{j\in M} \sigma x_{j,1}, \ldots, \bigcap_{j\in M} \sigma x_{j,m-1}, \sigma x_{j',m})), && \text{(Definition von } \eta(S\ \texttt{where}\ T = N)) \end{aligned}$$

   und jede einzelne Menge innerhalb der beiden Vereinigungen in dieser Zeile liegt offenbar in $\eta((x_{j,1}, \ldots, x_{j,m})\ [\texttt{where}\ T_j = N_j])$, also in $\eta(G)$. (Die Substitutionen, die für diesen Eintrag aus G für das eventuelle `where` benutzt werden, müssen gleich viel oder weniger Bedingungen genügen als die σ von oben.)

2. Es ist $\eta(G) = \eta(G') = \bigcup\{ \eta(F(a)) \mid a \text{ ist Eintrag in } \hat{G} \}$. Zu zeigen ist die Korrektheit von $\eta(F(a))$ für jeden Fall, der in der Definition von F vorkommt.

   - Fall (6.4): nichts ist zu zeigen.
   - Fall (6.5): Hier ist $C \leq a_p$, daraus folgt $\eta(C) \subseteq \eta(a_p)$, denn nach Voraussetzung ist a_p konventionell. Die Aussage des Lemmas folgt mit $\eta(\Pi_{\hat{p}}((a_1, \ldots, a_m))) = \underbrace{\eta(a_1) \times \cdots \times \eta(a_m)}_{\text{ohne } \eta(a_p)}$ und $\eta(a_1) \times \cdots \times \eta(a_m) \subseteq \eta(\hat{G}) = \eta(G)$.
   - Fall (6.6): Es ist $\eta(F((a_1, \ldots, a_m))\ \texttt{where}\ T = N) = \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} \eta(\sigma F((a_1, \ldots, a_m)))$. Für jedes einzelne σ ist Fall (6.5) anwendbar.
   - Fall (6.7): nichts zu zeigen.
   - Fall (6.9): Es ist $\eta(\Pi_{\hat{p}}((a_1, \ldots, a_m))\ \texttt{where}\ T^* = N^{**}) = \bigcup_{\sigma:\Lambda_p(\sigma T^*)\subseteq\eta(N^{**})} \eta(\sigma(\underbrace{a_1, \ldots, a_m}_{\text{ohne } a_p})) = \bigcup_{\sigma:\Lambda_p(\sigma T^*)\subseteq\eta(N^{**})} \underbrace{\eta(\sigma a_1) \times \cdots \times \eta(\sigma a_m)}_{\text{ohne } \eta(\sigma a_p)}$.

     Da N ohne Vereinigungen definiert ist, gilt $\eta(N) = \eta((\pi_1(N), \ldots, \pi_m(N)))$, wie leicht zu sehen ist — die primitiven Terme der einzelnen Komponenten von N hängen nicht von denen der anderen Komponenten ab.

Aus der Forderung $C \leq a_p$ `where` $T = N$ folgt $\eta(C) \subseteq \bigcup_{\sigma:\Lambda_p(\sigma T)\subseteq\eta(N)} \eta(\sigma a_p)$, denn nach Voraussetzung ist N konventionell, und damit weiter $\eta(C) \subseteq \bigcup_{\sigma:\Lambda_p(\sigma\Pi_{p_x}(T))\subseteq\eta(\pi_{p_x}(N))} \eta(\sigma a_p)$.
Für ein σ mit $\Lambda_p(\sigma T^*) \subseteq \eta(N^{**})$ kann man nun o.B.d.A. $\Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N))$ wählen (falls letzteres nicht möglich ist, muß $\eta(C)$ leer sein und es ist nichts zu zeigen). Mit $\eta(\sigma(a_1,\ldots,a_m)) \subseteq \eta(\hat{G})$ folgt die zu beweisende Aussage.

- Fall (6.8): Analog zu Fall (6.9) sind die Komponenten in $(a_1,\ldots,a_m)$ `where` $T = N$ unabhängig voneinander, also hat die gesamte Projektion $\Pi_{\hat{p}}((a_1,\ldots,a_m))$ die gewünschte Eigenschaft.
- Fall (6.10): Wir zeigen die Korrektheit der Terme in $\bigsqcup_i \Delta^i(\mathbf{0})(G')$ per Induktion über i. Das beweist die Korrektheit für konventionelle G'; wenn G und C aber konventionell sind, wie vorausgesetzt, so ist auch G' konventionell.
  Verankert wird mit $\eta_{\mathbf{0}}(G') = \emptyset$; hier ist nichts zu zeigen.
  Nun zum Induktionsschritt:
  Wir wissen: $\eta_{\Delta^{i+1}(\mathbf{0})}(G') = \bigcup \eta_{\Delta^i(\mathbf{0})}(F)$, wobei F für die einzelnen durch das Verfahren definierten Einträge steht. Wenn F durch einen der Fälle (6.4)–(6.9) entstanden ist, können wir wegen $\eta_{\Delta^i(\mathbf{0})}(F) \subseteq \eta(F)$ die bereits gezeigte Korrektheit von $\eta(F)$ benutzen.
  Für Fall (6.10) ist der Induktionsschritt noch zu zeigen:
  Es ist $\eta_{\Delta^i(\mathbf{0})}(\Pi_{\hat{p}}((a_1,\ldots,a_m))$ `where` $T^* = N^*) = \bigcup_{\sigma:\Lambda_p(\sigma T^*)\subseteq\eta_{\Delta^i(\mathbf{0})}(N^*)} \eta_{\Delta^i(\mathbf{0})}(\sigma(\underbrace{a_1,\ldots,a_m}_{\text{ohne } a_p}))$.
  Zu zeigen ist, daß jede der Mengen in der Vereinigung korrekt ist. Wir betrachten nun eine solche Menge mit passendem σ. Es gelte also:

$$\Lambda_p(\sigma T^*) \subseteq \eta_{\Delta^i(\mathbf{0})}(N^*)$$

  Da $\eta_{\Delta^i(\mathbf{0})}(N^*)$ nach Induktionsvoraussetzung korrekt ist, haben wir

$$\Lambda_p(\sigma T^*) \subseteq \{\, (\underbrace{y_1,\ldots,y_t}_{\text{ohne } y_{p_x}}) \in \pi_{\widehat{p_x}}(\eta(N)) \mid \forall y_{p_x} \in \eta(D) : (y_1,\ldots,y_t) \in \eta(N) \,\} \tag{6.13}$$

  Sei

$$S := \{\, \tau \in \Sigma_p \mid \tau_{|dom(\sigma)\setminus\{x\}} = \sigma_{|dom(\sigma)\setminus\{x\}} \wedge \Lambda_p(\tau x) \subseteq \eta(D') \,\}.$$

  S ist nicht leer, wenn die zweite Forderung für τ erfüllbar ist, bei konventionellem D' (bzw. D) also genau dann, wenn $\eta(D) \neq \emptyset$. Der Fall $\eta(D) = \emptyset$ sei zunächst zurückgestellt. Es gilt (6.13) somit auch, wenn man dort σ durch ein $\tau \in S$ ersetzt. Die Definition von S zieht auch

$$\forall \tau \in S : \Lambda_p(\tau(P(x))) \subseteq \eta(P(D')) = \eta(D)$$

  nach sich, und da T eine Ergänzung von T^* um $P(x)$ an der Stelle p_x ist, folgt mit ihr weiter:

$$\forall \tau \in S : \Lambda_p(\tau T) \subseteq \eta(N). \tag{6.14}$$

  Deshalb gilt:

$$\forall \tau \in S : \eta_{\Delta^i(\mathbf{0})}(\tau(a_1,\ldots,a_m)) \subseteq \eta((a_1,\ldots,a_m) \text{ `where` } T = N) \tag{6.15}$$

  An dieser Stelle benutzen wir $\eta(C) \subseteq \eta(a_p(x)$ `where` $x = D')$, wenn $\eta(D) \neq \emptyset$ (das war oben vorausgesetzt). Daraus und aus der Definition von S bekommen wir

$$\begin{aligned} \eta(C) &\subseteq \bigcup_{\tau':\Lambda_p(\tau' x)\subseteq\eta(D')} \eta(\tau' a_p) \\ &= \bigcup_{\tau\in S} \eta(\tau a_p), \end{aligned} \tag{6.16}$$

und aus (6.15) und (6.16) folgt nun für alle $\tau \in S$:

$$\begin{aligned}
\underbrace{\eta_{\Delta^i(\mathbf{0})}(\tau a_1) \times \cdots \times \eta_{\Delta^i(\mathbf{0})}(\tau a_m)}_{\eta(C) \text{ statt } \eta_{\Delta^i(\mathbf{0})}(\tau a_p)} &\subseteq \underbrace{\eta(\tau a_1) \times \cdots \times \eta(\tau a_m)}_{\eta(C) \text{ statt } \eta(\tau a_p)} \\
&\subseteq \eta(\tau a_1) \times \cdots \times \left(\bigcup_{\tau' \in S} \eta(\tau' a_p) \right) \times \cdots \times \eta(\tau a_m) \\
&= \bigcup_{\tau' \in S} (\eta(\tau' a_1) \times \cdots \times \eta(\tau' a_m)) \\
&\subseteq \eta((a_1, \ldots, a_m) \texttt{ where } T = N)
\end{aligned}$$

Auf alle außer der p-ten Stelle bezogen, heißt das

$$\begin{aligned}
&\forall \tau \in S : \eta_{\Delta^i(\mathbf{0})}\tau(\underbrace{a_1, \ldots, a_m}_{\text{ohne } a_p}) \\
&\quad \subseteq \{ (\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p}) \mid \forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \underbrace{\eta((a_1, \ldots, a_m) \texttt{ where } T = N)}_{\subseteq \eta(G)} \},
\end{aligned}$$

was den Beweis für diesen Fall abschließt, denn o.B.d.A. ist $\sigma \in S$ — der Eintrag in G' stellt ja keine Forderung an das Verhalten von σ auf x.

Bei $\eta(D) = \emptyset$ argumentieren wir folgendermaßen: Aus (6.13) ergibt sich, daß $\pi_{\widehat{p_x}}(\eta(N))$ und damit $\eta(N)$ nicht leer sein kann.[1][2] Da nun N konventionell ist (das folgt aus der gleichen Eigenschaft von G), muß es einen primitiven Kontext R geben mit $\Lambda_p(R) \subseteq \pi_{p_x}(\eta(N))$ (R ist die p_x-te Komponente des Kontextes, den Lemma 4.41 garantiert). Es sei deshalb hier $S := \{ \tau \in \Sigma_p \mid \tau_{|dom(\sigma)\setminus\{x\}} = \sigma_{|dom(\sigma)\setminus\{x\}} \wedge \tau x = R \}$, so daß sich (6.14) und weiter (6.15) ergeben. Wir beachten nun, daß sich die Formel (6.2) unter Kenntnis von $\eta(D) = \emptyset$ vereinfacht zu $\forall X \in \Lambda^{C,X} : \eta(C) \subseteq \eta(a_p(x) \texttt{ where } x = X)$. Man kann nun R für X einsetzen und erhält $\eta(C) \subseteq \eta(a_p(R)) = \eta(\tau a_p)$. Damit erhalten wir wieder die Beziehung (6.16) und schließlich die zu beweisende Aussage.

◇

Korollar 6.4.

$$\eta(f\text{-}C_2\text{-}C_1) \subseteq \{ (x_1, \ldots, x_n) \mid \forall x_{n+1} \in \eta(C_1) : (x_1, \ldots, x_{n+1}) \in \eta(f\text{-}C_2) \}.$$

Beweis. Für $\eta(C_1) \neq \emptyset$ ist das ein Spezialfall von Lemma 6.3. Bei $\eta(C_1) = \emptyset$ ist es trivial. ◇

Jetzt sind wir soweit, den Satz zur Korrektheit zu beweisen.

Beweis von Satz 6.2. Aus Korollar 6.4 folgt

$$\begin{aligned}
&\forall((x_1, \ldots, x_n), x_{n+1}) \in \eta(f\text{-}C_2\text{-}C_1, C_1) : (x_1, \ldots, x_{n+1}) \in \eta(f\text{-}C_2) \\
\Longrightarrow\ &\forall((y_1, \ldots, y_n), y_{n+1}) \in \gamma(f\text{-}C_2\text{-}C_1, C_1) : (y_1, \ldots, y_{n+1}) \in \gamma(f\text{-}C_2)
\end{aligned}$$

und, da jedes $y \in \gamma(f\text{-}C_2\text{-}C_1)$ nach Lemma 3.33 tatsächlich die Form $(y_1, \ldots, y_n)$ hat,

$$\Longrightarrow \gamma(f\text{-}C_2\text{-}C_1) \subseteq \{ (y_1, \ldots, y_n) \mid \forall y_{n+1} \in \gamma(C_1) : (y_1, \ldots, y_{n+1}) \in \gamma(f\text{-}C_2) \}.$$

Mit (6.12) folgt dann sofort die Aussage des Satzes. ◇

[1] Natürlich kann N leer sein, aber dann gibt es kein an dieser Stelle zu betrachtendes σ.

[2] Wenn N nur aus 1 Stelle besteht und T^* somit zu () degeneriert ist, funktioniert dieser Schluß nicht: Dieser Fall muß mit (6.7) oder (6.8) behandelt werden.

6.1.4 Vollständigkeit

Vollständigkeit des Verfahrens heißt:

$$\gamma(f\text{-}C_2\text{-}C_1) \supseteq \{\,(x_1,\ldots,x_n) \mid f(x_1,\ldots,x_n) \in \gamma(C_1 \overset{\forall}{\to} C_2)\,\} \tag{6.17}$$

sofern $f\text{-}C_2$ vollständig ist, d.h.

$$\gamma(f\text{-}C_2) \supseteq \{\,(x_1,\ldots,x_{n+1}) \mid f(x_1,\ldots,x_n)\ x_{n+1} \in \gamma(C_2)\,\} \tag{6.18}$$

Wir beweisen für Analysen ohne `where`-Rekursion eine entsprechende Beziehung, die mit η formuliert ist. In sehr vielen Fällen läßt sich sogar die echte Vollständigkeit zeigen.

Satz 6.5. *Es gelte* (6.18)*. Die Analyse benutze nicht Fall* (6.10)*.*
Wenn $f\text{-}C_2$ konventionell ist, gilt

$$\eta(f\text{-}C_2\text{-}C_1) \supseteq \{\,(x_1,\ldots,x_n) \in \Lambda_p \mid \forall x_{n+1} \in \eta(C_1) : f(x_1,\ldots,x_n)\ x_{n+1} \in \gamma(C_2)\,\}.$$

Wenn C_1 konventionell oder inkonsistent ist, oder wenn $f\text{-}C_2$ konventionell ist und C_1 nur aus `Bot` *und Konstruktoren besteht, gilt sogar*

$$\gamma(f\text{-}C_2\text{-}C_1) \supseteq \{\,(x_1,\ldots,x_n) \mid \forall x_{n+1} \in \eta(C_1) : f(x_1,\ldots,x_n)\ x_{n+1} \in \gamma(C_2)\,\}.$$

Damit wird unter Benutzung von Lemma 4.37 die echte Vollständigkeit (6.17) *impliziert.*

Für den Beweis benötigen wir noch ein Lemma und ein Korollar, in denen es um die Vollständigkeit des Verfahrens im zweiten Schritt und für primitive Terme geht. Wir setzen dabei wieder voraus, daß die allgemeine `where`-Rekursion nicht vorkommt.

Zunächst jedoch beweisen wir dafür zwei Hilfssätze: einen für den Fall, daß C aus `Bot` und Konstruktoren besteht, den anderen für allgemein konventionelle C.

Lemma 6.6. *C bestehe aus Konstruktoren und* `Bot`*, und G sei ein konventioneller Kontext. Es gelte*

$$(x_1,\ldots,x_m) \in \eta(G) \quad \wedge \quad x_p \in \eta(C).$$

Dann gilt

$$(\underbrace{x_1,\ldots,x_m}_{\text{ohne } x_p}) \in \eta((\underbrace{y_1,\ldots,y_m}_{\text{ohne } y_p})\ \texttt{where}\ (\underbrace{y_1,\ldots,y_m}_{C \text{ statt } y_p}) = G).$$

Beweis. Wegen $(x_1,\ldots,x_m) \in \eta(G) \subseteq \Lambda_p$ haben wir $\forall i : \exists! P_i \in \Lambda_p^C : x_i \in \Lambda_p(P_i)$. Dabei ist offenbar $P_p = C$. Nach Lemma 4.41 gilt hiermit $\Lambda_p(P_1,\ldots,P_m) \subseteq \eta(G)$. Die Substitution $\sigma : y_i \mapsto P_i$ erfüllt also $\Lambda_p(\sigma(\underbrace{y_1,\ldots,y_m}_{C \text{ statt } y_p})) \subseteq \eta(G)$ und bezeugt damit die Wahrheit der zu beweisenden Aussage. ◇

Lemma 6.7. *Sei C konventionell und seien $(\underbrace{x_1,\ldots,x_m}_{\text{ohne } x_p})$ gegeben mit*

$$\forall x_p \in \eta(C) : (x_1,\ldots,x_m) \in \gamma(G).$$

Dann gilt für alle $x_p \in \eta(C)$:

$$(x_1,\ldots,x_m) \equiv_c \bigsqcup_i^c (x_1^i,\ldots,x_m^i) \text{ mit } \forall i : (x_1^i,\ldots,x_m^i) \in \eta(G)$$

und mit o.B.d.A. allen $x_p^i \in \eta(C)$.

Beweis. Die erste Folgerung ist die ausgeschriebene Definition von γ (unter Verwendung des hier oft benutzten Lemmas 3.33). Daß o.B.d.A. die $x_p^i \in \eta(C)$ sein dürfen, sagt Lemma 4.45, denn es ist $\bigsqcup_i^c x_p^i \equiv_c x_p$. ◇

Diese beiden Lemmata werden nun vom folgenden verwendet.

Lemma 6.8. *Sei G' der Name des Analyseergebnisses von Schritt 2 des Verfahrens bezüglich G, C, p. Wenn die Analyse* nicht *Fall* (6.10) *benutzt, gilt:*

$$\eta(G') \supseteq \{ (\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p}) \in \pi_{\hat{p}}(\eta(G)) \,|\, \forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \eta(G) \} \tag{6.19}$$

Falls C nur aus `Bot` *und Konstruktoren besteht, gilt die Aussage unter der Bedingung, daß G konventionell ist. In diesem Fall, und auch wenn C ein sonstiger konventioneller Kontext ist, gilt dann aber sogar*

$$\gamma(G') \supseteq \{ (\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p}) \,|\, \forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \gamma(G) \}. \tag{6.20}$$

Beweis.

- Falls C nur aus Konstruktoren und `Bot` besteht, betrachten wir ein Jec-Konstrukt $(\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p})$ mit

  $$\forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \gamma(G).$$

  Für alle $x_p \in \eta(C)$ ist somit $(x_1, \ldots, x_m) \equiv_c \bigsqcup_i^c (x_1^i, \ldots, x_m^i)$ mit $\forall i : (x_1^i, \ldots, x_m^i) \in \eta(G)$. Wegen Lemma 6.7 sind o.B.d.A. alle $x_p^i \in \eta(C)$. Weil G konventionell ist, folgt mit Lemma 6.6 für alle $x_p \in \eta(C)$:

  $$(\underbrace{x_1^i, \ldots, x_m^i}_{\text{ohne } x_p^i}) \in \eta((\underbrace{y_1, \ldots, y_m}_{\text{ohne } y_p}) \texttt{ where } (\underbrace{y_1, \ldots, y_m}_{C \text{ statt } y_p}) = G).$$

  Die Quantifizierung „Für alle $x_p \in \eta(C)$“ ist dabei wirkungslos, da x_p in der Aussage nicht vorkommt und $\eta(C)$ nicht leer ist, kann also gestrichen werden. Bildet man auf jeder Seite die clubs, ergibt sich (6.20).

  Auch (6.19) gilt, da es eine direkte Folge von (6.20) ist.

- Wenn C nicht nur aus `Bot` und Konstruktoren aufgebaut ist, sei $(\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p})$ gegeben mit

  $$\forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \eta(G). \tag{6.21}$$

  Es existieren somit $r \geq 1$ Einträge $g_1, \ldots, g_r$ aus G, so daß

  $$\forall x_p \in \eta(C) : \exists i : (x_1, \ldots, x_m) \in \eta(g_i). \tag{6.22}$$

  Dabei sei die Menge der g_i minimal, so daß die Formel noch erfüllt ist.

  Entsprechend der Indizes, die in der Beschreibung des Analyseverfahrens verwendet werden, sei $g_i = a_{j_i} = (a_{j_i,1}, \ldots, a_{j_i,m})$ [`where` $T_{j_i} = N_{j_i}$].

  Falls für eines der $x_p \in \eta(C)$ mit zugehörigem g_i gilt $\eta(C) \subseteq \eta(a_{j_i,p}$ [`where` $T_{j_i} = N_{j_i}$]), so übernimmt die Analyse die entsprechenden Komponenten von g_i direkt ins Ergebnis, es ist $(\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p}) \in \eta(G')$ (sofern g_i nicht Fall (6.10) entspricht, was wir ausgeschlossen haben), also schon die zu beweisende Aussage.

  Ist das nicht der Fall, gilt also für jedes g_i: $\eta(C) \not\subseteq \eta(a_{j_i,p}$ [`where` $T_{j_i} = N_{j_i}$]). Damit sind alle ihre Indizes bei der Analyse in die Indexmenge I aufgenommen worden. Die Menge dieser Indizes j_i, $1 \leq i \leq r$, heiße M. Wegen der Minimalität der Menge der g_i folgt aus der Formel (6.22) $\forall i : (\underbrace{x_1, \ldots, x_m}_{\text{ohne } x_p}) \in \pi_{\hat{p}}(\eta(g_i))$. Damit gilt: $\forall k$ mit $k \neq p : x_k \in \bigcap_{j \in M} \pi_k(\eta(a_j))$.

  Weiterhin ist ebenfalls wegen (6.22) $\forall x_p \in \eta(C) : x_p \in \bigcup_{j \in M} \pi_p(\eta(a_j))$.

$\hat{g} = (\bigcap_{j\in M} a_{j,1}, \ldots, \bigcap_{j\in M} a_{j,p-1}, \langle a_{j,p}\rangle_{j\in M}, \bigcap_{j\in M} a_{j,p+1}, \ldots, \bigcap_{j\in M} a_{j,m})$ [`where` $T = N$], ein Eintrag in der Vereinigung $\hat{G}$, erfüllt also die folgende Beziehung:

$$\forall x_p \in \eta(C) : (x_1, \ldots, x_m) \in \eta(\hat{g}). \tag{6.23}$$

Jetzt ist noch zu zeigen: $\underbrace{(x_1, \ldots, x_m)}_{\text{ohne } p} \in \eta(F(\hat{g}))$, wobei F die Transformationsfunktion der Analyse ist. Wir stellen zuerst eine Implikation von (6.23) fest: $\eta(C) \subseteq \pi_p(\eta(\hat{g}))$ und deshalb $\eta(C) \subseteq \eta(\hat{g}_p$ [`where` $T = N$]), wobei $\hat{g}_p = \langle a_{j,p}\rangle_{j\in M}$ die p-te Komponente des Jec in $\hat{g}$ ist.

- $\hat{g}$ kann deswegen nicht von den Fällen (6.4) oder (6.7) behandelt werden.
- Fällt $\hat{g}$ unter (6.5), kommt gerade $\underbrace{(x_1, \ldots, x_m)}_{\text{ohne } x_p}$ in das Ergebnis $\eta(F(\hat{g}))$, wie gewünscht.
  Auch in den Fällen (6.6) und (6.8) werden diese Komponenten des Jecs $\hat{g}$ in das Ergebnis übernommen.
- Das ist schließlich auch in Fall (6.9) so, denn wenn auch T und N zu T^* und N^{**} werden, unterscheiden sich die Substitutionen, die das `where` vor und nach der Anwendung von F betrachtet (nämlich die σ_1 mit $\Lambda_p(\sigma_1 T) \subseteq \eta(N)$ und die σ_2 mit $\Lambda_p(\sigma_2 T^*) \subseteq \eta(N^{**})$) nur auf den Variablen, welche in T^* bezüglich T fehlen, und jene kommen in $\hat{g}$ nicht außerhalb der p-ten Komponente vor.

- Ist C konventionell, so beweist der vorige Punkt zunächst unabhängig von dieser Eigenschaft die Aussage (6.19). Wendet man nun Lemma 6.7 darauf an, ergibt sich (6.20).

◇

Korollar 6.9. *Wenn Fall* (6.10) *nicht für die Analyse benötigt wird, ist*

$$\eta(f\text{-}C_2\text{-}C_1) \supseteq \{\, (x_1, \ldots, x_n) \,|\, \forall x_{n+1} \in \eta(C_1) : (x_1, \ldots, x_{n+1}) \in \eta(f\text{-}C_2) \,\}.$$

Besteht C_1 nur aus `Bot` *und Konstruktoren und ist f-C_2 konventionell, oder ist C_1 konventionell oder inkonsistent, gilt sogar*

$$\gamma(f\text{-}C_2\text{-}C_1) \supseteq \{\, (x_1, \ldots, x_n) \,|\, \forall x_{n+1} \in \eta(C_1) : (x_1, \ldots, x_{n+1}) \in \gamma(f\text{-}C_2) \,\}.$$

Beweis. Für $\eta(C_1) \neq \emptyset$ ist das ist ein Spezialfall von Lemma 6.8; die Beschränkung auf $(x_1, \ldots, x_n) \in \pi_{\widehat{n+1}}(\eta(f\text{-}C_2))$, die in (6.19) in jenem Lemma gefordert wird, ergibt sich in diesem Fall automatisch. Falls $\eta(C_1) = \emptyset$, ist die Aussage trivial. ◇

Nun endlich kann der Satz zur Vollständigkeit bewiesen werden.

Beweis von Satz 6.5. Aus (6.18) kann man unter Gebrauch von Lemma 4.43

$$\eta(f\text{-}C_2) \supseteq \{\, (x_1, \ldots, x_{n+1}) \in \Lambda_p \,|\, f(x_1, \ldots, x_n)\ x_{n+1} \in \gamma(C_2) \,\}. \tag{6.24}$$

folgern, sofern f-C_2 konventionell ist.
Mit dieser Beziehung bzw. mit (6.18) direkt ergibt sich aus Korollar 6.9 die Aussage des Satzes. ◇

6.2 Analyse von $f \in C_1 \xrightarrow{\exists} C_2$

Die Analyse von $f \in C_1 \xrightarrow{\exists} C_2$ kann durch eine Projektion gelöst werden. Wie im allgemeinen Fall der $C_1 \xrightarrow{\forall} C_2$-Analyse wird zuerst der Lösungskontext für $f\ y \in C_2$ berechnet. Während bei der $C_1 \xrightarrow{\forall} C_2$-Analyse dann die rechteste Komponente des Zwischenresultats f-C_2 auf das Enthaltensein von C_1 überprüft werden muß, reicht es hier aus, auf das Vorkommen eines beliebigen Elementes

der Konkretisierung von C_1 in der rechtesten Komponente von $f\text{-}C_2$ zu testen. Das wird mit einem Schnitt und einer Projektion erreicht.

Dabei wird die Projektion $\pi_{\widehat{n+1}}$ auf einen Schnittkontext angewendet, was nach der in dieser Arbeit gegebenen Definition nicht definiert ist. Wir gehen deshalb in diesem Abschnitt davon aus, daß es eine Projektion gibt, die auch auf Schnitten definiert ist und Lemma 4.54 erfüllt, und benutzen den gleichen Namen dafür wie für die in Abschnitt 4.7.4 explizit angegebene Projektion.

Verfahren 6.3 (Analyse von $f \in C_1 \overset{\exists}{\to} C_2$).
f habe n Variablen.

Falls $\eta(C_1) = \emptyset$, ist

$$f\text{-}C_2\text{-}C_1 = \langle\rangle$$

die Lösung und es muß nichts mehr gemacht werden.

Sonst analysiere $f\ y \in C_2$. Das Ergebnis wird unter dem Namen $f\text{-}C_2$ in die Menge der Kontextdefinitionen aufgenommen.

Das Endergebnis ist

$$f\text{-}C_2\text{-}C_1 = \pi_{\widehat{n+1}}(f\text{-}C_2 \cap (\mathtt{Top}, \ldots, \mathtt{Top}, C_1)).$$

6.2.1 Korrektheit und Vollständigkeit des Verfahrens

Die Analyse rechnet korrekt bzw. vollständig, wenn

$$\gamma(f\text{-}C_2\text{-}C_1) \subseteq \{\, (x_1, \ldots, x_n) \mid f(x_1, \ldots, x_n) \in \gamma(C_1 \overset{\exists}{\to} C_2) \,\} \tag{6.25}$$

sofern $f\text{-}C_2$ korrekt ist, d.h.

$$\gamma(f\text{-}C_2) \subseteq \{\, (x_1, \ldots, x_{n+1}) \mid f(x_1, \ldots, x_n)\ x_{n+1} \in \gamma(C_2) \,\} \tag{6.12}$$

beziehungsweise

$$\gamma(f\text{-}C_2\text{-}C_1) \supseteq \{\, (x_1, \ldots, x_n) \mid f(x_1, \ldots, x_n) \in \gamma(C_1 \overset{\exists}{\to} C_2) \,\} \tag{6.26}$$

sofern $f\text{-}C_2$ vollständig ist, d.h.

$$\gamma(f\text{-}C_2) \supseteq \{\, (x_1, \ldots, x_{n+1}) \mid f(x_1, \ldots, x_n)\ x_{n+1} \in \gamma(C_2) \,\} \tag{6.18}$$

Wir wissen durch die Projektionseigenschaft von $\pi_{\widehat{n+1}}$ (Lemma 4.54):

$$\begin{aligned} \eta(f\text{-}C_2\text{-}C_1) &= \pi_{\widehat{n+1}}(\eta(f\text{-}C_2 \cap (\mathtt{Top}, \ldots, \mathtt{Top}, C_1))) \\ &= \{\, (x_1, \ldots, x_n) \mid \exists x_{n+1} : (x_1, \ldots, x_{n+1}) \in \eta(f\text{-}C_2) \,\} \end{aligned} \tag{6.27}$$

Der Fall $\eta(C_1) = \emptyset$ führt in der obigen Gleichung zu $\eta(f\text{-}C_2\text{-}C_1) = \emptyset$, genau wie es bei der Analyse tatsächlich gilt, und muß deswegen nicht gesondert betrachtet werden.

Die folgenden beiden Sätze können bewiesen werden:

Satz 6.10. *Es gelte* (6.12). *Dann ist*

$$\eta(f\text{-}C_2\text{-}C_1) \subseteq \{\, (x_1, \ldots, x_n) \mid \exists x_{n+1} \in \eta(C_1) : f(x_1, \ldots, x_n)\ x_{n+1} \in \gamma(C_2) \,\}.$$

Beweis. Die Aussage gewinnt man aus (6.27) unter Benutzung von $\eta(f\text{-}C_2) \subseteq \gamma(f\text{-}C_2)$ und (6.12). ◇

Satz 6.11. *Es gelte* (6.18), *und $f\text{-}C_2$ sei konventionell. Dann ist*

$$\eta(f\text{-}C_2\text{-}C_1) \supseteq \{\, (x_1, \ldots, x_n) \in \Lambda_p \mid \exists x_{n+1} \in \eta(C_1) : f(x_1, \ldots, x_n)\ x_{n+1} \in \gamma(C_2) \,\}.$$

Beweis. Wenn $f\text{-}C_2$ konventionell ist, kann man aus (6.18) wie schon bei der $C_1 \overset{\forall}{\to} C_2$-Analyse auf Seite 74

$$\eta(f\text{-}C_2) \supseteq \{\, (x_1, \ldots, x_{n+1}) \in \Lambda_p \mid f(x_1, \ldots, x_n)\ x_{n+1} \in \gamma(C_2) \,\} \tag{6.24}$$

folgern. Mit dieser Ungleichung kann aus (6.27) die Aussage des Lemmas abgeleitet werden. ◇

Kapitel 7

Implementierung

In diesem Kapitel wird die Umsetzung des theoretischen Teils in Haskell beschrieben. Alle erwähnten Programmzeilen sind durch Schreibmaschinenschrift hervorgehoben und beginnen mit einem > (zur Erläuterung s. Abschnitt 2.4). Fehlt dieses Zeichen, so wird über Programmcode gesprochen, der nicht in der wirklichen Implementierung vorkommt oder zumindest nicht an der jeweiligen Stelle.

Eine praktische Umsetzung der Kontextanalyse von Marko Schütz ([Sch99a]) lag schon zu Beginn dieser Arbeit vor. Diese unterlag jedoch beständiger Entwicklung durch den Autor, so daß Teile des Codes in die vorliegende Arbeit übernommen wurden, aber auch einiges im Bereich der (konventionellen) Kontextanalyse von mir selbst geschrieben wurde. Übernommene Teile werden hier als solche kenntlich gemacht.

Das Kapitel ist wie folgt gegliedert:

Die Abschnitte 7.1 und 7.2 behandeln mit Kontexten und der Kernsprache die Datenkonstrukte, auf denen die Analyse arbeitet. Im Fall der Kontexte wird dabei ausführlich zum einen auf die bereitgestellten Funktionen, zum anderen auf die verwendeten Programmierstilmittel eingegangen.

In den Abschnitten 7.3 bis 7.5 geht es um die Implementierung der Tableaus und der Abläufe darauf. Dieses Rüstzeug stammt aus [Man99]. Wir beschreiben es zunächst allgemein und überlegen dann, wie die Kontextanalyse dieses Werkzeug nutzen kann.

Die Abschnitte 7.6 und 7.7 dokumentieren die Umsetzung der Analysen aus Kapitel 6 und die Erzeugung eines Kontextes aus dem Tableau.

Schließlich geht es in Abschnitt 7.8 um Code, der noch nicht erklärt wurde, aber aus verschiedenen Gründen noch dargestellt werden soll.

Von nahezu allen Modulen genutzt werden drei einfache Typsynonyme, die zur besseren Lesbarkeit und Mnemonik beitragen sollen und mit denen wir hiermit die Vorstellung des Codes beginnen:

```
> type Name  = String
> type Tag   = Int
> type Arity = Int
```

„Tag“ und „Arity“ sind die englischen Wörter für „Bezeichner“ und „Stelligkeit“. Diese beiden Angaben sind genau die zur Bezeichnung eines Konstruktors benötigten.

7.1 Kontexte

7.1.1 Darstellung

Kontexte sind durch einen algebraischen Datentyp mit dem Namen `DemandTerm` (englisch Demand: Anforderung) repräsentiert. Die Gesamtstruktur und die Konstruktornamen lehnen sich an

den entsprechenden Code in [Sch99a] an. Jede Möglichkeit aus der Hauptregel in Abbildung 4.1 ist vorgesehen; zusätzlich die neu eingeführten Funktionskontexte. `where`-Kontexte stehen weiter unten als letzte in der Auflistung, damit sie in geordneten Listen zur besseren Übersicht von der `Ord`-Instanz von `DemandTerm` hinter allen anderen aufgeführt wird.

```
> data DemandTerm
>   = DBot
>   | DFun
>   | DName Name
>   | DConstr Tag Arity [DemandTerm]
>   | DUnion (Set DemandTerm)
>   | DIntersection (Set DemandTerm)
>   | DForall DemandTerm DemandTerm
>   | DExists DemandTerm DemandTerm
```

Für die Typen der Argumente von `DUnion` und `DIntersection` wurden Mengen von Kontexten gewählt, basierend auf einer geordneten Listendarstellung ohne Duplikate (s. Abschnitt 7.8.2). Dies vermeidet schon im Ansatz die eventuelle Bildung von Redundanz durch mehrfache Vorkommen gleicher Kontexte in einem Schnitt oder einer Vereinigung — das könnte nämlich leicht zu überflüssigen Berechnungen führen.

Hinzu kommt eine Darstellung für Variablen; außerdem bekommen auch Jecs einen eigenen Konstruktor, damit keine Konflikte mit gewöhnlichen Konstruktoren entstehen:

```
>   | DVar Name
>   | DJec Arity [DemandTerm]
```

Schließlich ermöglichen wir es, Constraints (s.u.) in einen `DemandTerm` zu verpacken, vor dem noch fehlenden Konstruktor für `where`-Ausdrücke.

```
>   | DConstraints [Constraint]
>   | DemandTerm `DWhere` (DemandTerm,DemandTerm)
>   deriving (Eq,Ord)
```

Die Darstellung von S `where` $T = N$ ist hier `S `DWhere` (T,N)`.

Der Hintergrund von `DConstraints` ist die Definition neuer Kontextnamen für die Ergebnisse von Tableauanalysen: Diese Namen werden naheliegenderweise aus den Constraints der entsprechenden Anfrage zusammengestellt. Speichert man diese Constraints nun nicht als `DName` x, wobei x eine wie auch immer geartete Zeichenkette ist, sondern direkt unter dem Konstruktor `DConstraints`, so kann die Regel „Unabhängige Constraints“ direkt auf die Struktur dieser Constraints zugreifen, ohne eine Zeichenkette parsen zu müssen.

`DConstraints` ist also nur eine zweite Darstellung von Kontextnamen. Dieser Konstruktor und `DName` werden im Prinzip überall gleich behandelt.

Der Typ `Constraint` ist eine direkte Umsetzung der theoretischen Constraints und wurde aus dem Code von Marko Schütz übernommen. Gleiches gilt für einen eigenen Typ für CCVs, dessen Verwendung sich deswegen empfiehlt, weil Constraints und CCVs in einem Tableau ohnehin wegen der Schleifenregeln unterschieden werden müssen.

```
> data Constraint
>   = (:<-:) { expr :: LambdaCExpr, demand :: DemandTerm }
```

```
>   deriving (Eq,Ord)
> data CCV = In {var :: LambdaCExpr, c :: DemandTerm}
>   deriving Eq
```

Der für Λ-Ausdrücke stehende Typ `LambdaCExpr` wird weiter unten beschrieben.

Wie wir oben gesehen haben, stehen zwei Formen für Kontextdefinitionen bereit: `DName Name` und `DConstraints [Constraint]`. Genauer gesagt repräsentieren sie offenbar nur die *Namen* von Kontexten. Wo sind nun deren Definitionen? Sie müssen in einer Tabelle nachgeschlagen werden. Diese Implementierung von $\mathcal{C}$ benutzt eine Zustandsmonade. Worum es sich bei monadischem Programmierstil handelt, soll zunächst ein eigener Abschnitt erörtern.

7.1.2 Monadisches Programmieren

In dieser Arbeit wird ein Programmierstil verwendet, den man als *monadisches Programmieren* bezeichnet und der spätestens seit Einführung des monadischen Ein-/Ausgabe-Konzeptes in Haskell 1.3 ([HAB+96]) zu den verbreiteten Haskell-Programmiertechniken gehört. Da das Thema nicht unbedingt leicht zugänglich ist, wird hier eine kurze Einführung gegeben. Die formale Definition stützt sich auf [Wad95], ebenso ist die Sichtweise einer Monade als Berechnung aus dieser Quelle entnommen.

Eine *Monade* kann man als Verpackung von Datenobjekten betrachten. Die monadischen Operationen können ein Objekt beliebigen Typs verpacken und ein bereits verpacktes Objekt weiterverarbeiten zu einem beliebigen anderen Objekt, wobei die Verpackung nicht entfernt wird.

Das Innere der Verpackung weist eine bestimmte Struktur auf, die das Wesen der jeweiligen Monade ausmacht. Nur eine Möglichkeit, aber vielleicht die am meisten verwendete, ist die Struktur von *Zustandsmonaden*, die auch in dieser Arbeit vorkommen. Dort befindet sich ein Zustand innerhalb der Verpackung, der bearbeitet werden und mit dem verpackten Datenobjekt interagieren kann. Der Zustand hat eine feste, aber beliebige Form. Im Sonderfall der in Haskell eingebauten `IO`-Monade ist es sogar die gesamte „Außenwelt", von der innerhalb der Verpackung gelesen oder in die geschrieben werden kann.

Der hauptsächliche Gewinn, den man durch die Verwendung von Monaden erzielt, besteht in der bequemen Weitergabe bestimmter Daten von Funktion zu Funktion, nämlich ohne daß diese selbst erwähnt werden mußsen. Eine Zustandsmonade wird so zur Simulation eines globalen Zustandes. Als einen Nachteil kann man empfinden, daß eine solche monadische Funktion einen neuen Typ erhält, der sie explizit als monadisch ausweist. Sie kann damit nur von Funktionen aufgerufen werden, die selbst monadisch sind, oder die ein Objekt „von Hand" auspacken (wenn der Zustand in der Monade — bzw. die jeweilige Struktur — nicht mehr in dieser Form gebraucht wird).

Eine Monade wird in Haskell von einem algebraischen Datentyp M gebildet, auf dem zwei polymorphe Funktionen `return` und `(>>=)` aus der Haskell-Standardklasse `Monad` definiert sind.

$$\begin{aligned} \texttt{return} &:: a \to M\ a \\ \texttt{(>>=)} &:: M\ a \to (a\ \to M\ b) \to M\ b \end{aligned}$$

`return` verpackt also ein Objekt in eine Monade. `(>>=)` verarbeitet das Objekt in der Monade seines ersten Arguments mit Hilfe der Funktion im zweiten Argument; jenes ist eine Funktion, die das Objekt als Parameter nimmt und wieder verpackt. Die oben erwähnte Struktur der Monade entspricht der Art und Weise, wie die beiden Monadenoperationen konkret implementiert sind.

Diese Implementierung kann nicht beliebig sein, sondern muß bestimmte Gesetze erfüllen. Das theoretische Konzept, das in den Monaden steckt, kann nämlich nur greifen, wenn die folgenden Bedingungen erfüllt sind (das „=" ist die semantische Gleichheit):

- Linkes Einselement:

  ```
  return e >>= k  =  k e
  ```

- Rechtes Einselement:

```
m >>= return = m
```

- Assoziativität:

```
m >>= (\a -> (n >>= k)) = (m >>= \a -> n) >>= k
```

(Die letzte Bedingung wird wegen der Überdeckung von Namensbereichen nur für den Fall gefordert, daß `a` nicht frei in `k` vorkommt.)

Diese Bedingungen sichern, daß das folgende Bild wirklich zutrifft: Man kann sich einen Ausdruck vom Typ $M\ a$ als eine Berechnung vorstellen, die möglicherweise innerhalb der Monadenverpackung etwas ändert (bei Zustandsmonaden den Zustand) und dann ein Objekt vom Typ b zurückgibt. `return` x ändert nichts, sondern packt nur x ein. m `>>=` f entspricht einer Sequentialisierung der Berechnungen von m und $f\ y$, wobei das aus m resultierende Objekt y an f weitergereicht wird.

Eine Zustandsmonade aus [Jon95] diene uns als erläuterndes Beispiel:

```
data State s a = ST (s -> (a,s))

instance Monad (State s) where
  return x = ST (\s -> (x,s))
  m >>= f  = ST (\s0 -> let ST m'  = m
                            (x,s1) = m' s0
                            ST f'  = f x
                        in  (y,s2))
```

`s` ist der Zustand, dessen Typ hier durch Polymorphie frei wählbar ist. Der Monadentyp ist im wesentlichen eine Funktion vom Typ `(s -> (a,s))`. Was hat das zu bedeuten? Nun, man sieht hier sehr schön die Auffassung bestätigt, daß der Zustand geändert wird (denn ein solcher wird sowohl als Argument genommen als auch zurückgegeben) und daß außerdem ein Objekt vom Typ `a` zurückgegeben wird. Man sieht auch, daß `return x` eine Verpackung von `x` darstellt, ohne den Zustand zu ändern. Bei der Funktion `(>>=)` kann man erkennen, daß `m` sowohl seinen Zustand `s1` als auch sein Objekt `x` and `f` weiterreicht, was einer Hintereinanderkoppelung von `m` und `f` entspricht.

Eine konkreter Ausdruck vom Monadentyp ist z.B. `ST (\x -> (x, x+1))` für den Fall, daß der Zustand eine Zahl ist. Dieser Ausdruck entspricht einer Berechnung, die die Zahl im Zustand zurückgibt und sie im Zustand gleichzeitig erhöht.

Die Ausführung einer solchen monadischen Berechnung, die ja im allgemeinen durch Verwendung von `>>=` auch aus vielen hintereinandergeschalteten Komponenten bestehen kann, wird erst dann begonnen, wenn der Ausdruck auf einen Anfangszustand angewendet wird, so daß die ganze Berechnungskette nun darauf „arbeiten“ kann.

Wie oben erwähnt, können Zustandsmonaden verwendet werden, um einen globalen Zustand zu simulieren. Ohne Verwendung einer Monade müßten alle Operationen auf den Werten, die sich im Zustand befinden, mit ebendiesen als Parameter versorgt werden. Die Monade erlaubt es dagegen, solche Operationen einmal monadisch zu definieren und bei deren Aufruf auf die Angabe der einzelnen Werte zu verzichten.

In unserem Programm ist die Menge der Kontextdefinitionen ein guter Kandidat dafür, von einer Monade verpackt zu werden, denn es müssen oft Kontexte gelesen und mitunter auch neue definiert werden. Dies wird im folgenden Abschnitt angegangen.

7.1.3 Die Monade für Kontextdefinitionen

Eine Zustandsmonade wird die Kontextdefinitionen verwalten. Dafür legen wir zunächst Paare als Datentyp für beide Formen von Definitionen (Namen und Constraints) fest:

```
> type DemandDef a = (a, DemandTerm)
> type DNameDef = DemandDef Name
> type DNameEnv = [DNameDef]
> type DConDef = DemandDef [Constraint]
> type DConEnv = [DConDef]
```

Der Zustand, also die Menge der Kontextdefinitionen, ist nun ein algebraischer Datentyp, der die Listen beider Formen von Definitionen enthält. Auch eine Datenstruktur für einzelne Elemente dieser Listen wird erklärt.

```
> data DState = DState { dNameEnv :: DNameEnv,
>                       dConEnv :: DConEnv,
>                       dTempNameEnv :: DNameEnv }
>   deriving (Show, Eq)
> data DStateElem = DNameDefElem     DNameDef
>                 | DTempNameDefElem DNameDef
>                 | DConDefElem      DConDef
>   deriving (Show)
```

Die dritte Komponente `dTempNameEnv` des Zustands wird nur im Einzelfall benutzt, um vorübergehende Definitionen aufzunehmen, die nicht mit denen in `dNameEnv` interferieren sollen. Im Normalfall ist sie eine leere Liste.

Die Datenstruktur der eigentlichen Monade soll nun die Transformationen auf dem Zustand repräsentieren. Benötigt werden Operationen zum

- **Lesen:** Eine Lesefunktion bekommt den Zustand als Argument mit.
- **Schreiben:** Die einzig nötige Art des Schreibens in den Zustand ist das Anhängen einer neuen Definition. Soll der Zustand einmal gelöscht werden, so ist das mit dem Anwenden einer monadischen Funktion auf einen Zustand zu erreichen (der zurückgegebene Zustand kann ignoriert werden) und der anschließenden Bildung eines neuen monadischen Ausdrucks.

  Für das Schreiben einer neuen Definition ist nur die Angabe der Definition selbst erforderlich.
- Um beide Möglichkeiten auf eine Basis zu bringen und damit Verkettungen der Operationen zu ermöglichen, braucht man noch eine allgemeine **Transformation.** Deren Funktionalität setzt sich aus denen der beiden vorigen Punkte zusammen: Der Zustand muß als Argument verfügbar sein, und zurückgegeben werden muß eine Liste von anzuhängenden Kontextdefinitionen.

Der Vollständigkeit halber fügen wir noch eine Möglichkeit hinzu, die den Zustand gänzlich **ignoriert.**

Das Ganze wird ein algebraischer Datentyp. Der Name `DDAction` steht für „Action on demand definitions“.

```
> data DDAction a = Transformer (DState -> (a, [DStateElem]))
>                 | Reader      (DState ->  a)
>                 | Appender    (a,DStateElem)
>                 | Ignorer     a
```

Der `Transformer`-Part alleine würde für die Funktionalität der Monade ausreichen. Genau dafür wurde er ja auch angelegt! Für die Einbindung der anderen Teile spricht jedoch, daß Informationen über die Nutzung des Zustands dann besser erkannt werden können, und zwar sowohl

vom Compiler/Interpreter als auch vom Menschen. Ersterer kann Code erzeugen, bei dessen Ablauf z.B. der Folgezustand eines `Readers` direkt ermittelt werden kann (er ist nämlich gleich dem Ausgangszustand), ohne daß die entsprechende monadische Funktion weiter ausgewertet werden muß. Der Programmierer kann leichter erkennen, wie sich eine monadische Funktion verhält.

Im Prinzip handelt es sich um vier einzelne Monaden, die in einem Typ integriert sind. Diese können nun nicht vom Typ her unterschieden werden, was durchaus sinnvoll wäre, um in der Programmierung mehr Klarheit zu schaffen. Jedoch sollen Anwendungen die verschiedenen Monaden miteinander mit der Funktion >>= aus der Standard-Monadenklasse von Haskell kombinieren können; deren Argumente müssen aber auf Monaden gleichen Typs basieren. Zwei Lösungen wären denkbar:

- Jede im Code vorkommende Monade, die kein `Transformer`-Konstrukt ist, müßte mittels einer Umwandlungsfunktion zu einer solchen gemacht werden. Im hier realisierten Ansatz genügt es dagegen, diese Umwandlung nur innerhalb der Definition von >>= zu erwähnen.
- Ein eigenes >>=, das polymorph über den speziell verwendeten Monadentypen ist, ließe sich definieren. Haskell-Ausdrücke in `do`-Notation für Operationen auf diesen Monaden bleiben jedoch davon unbeeinflußt, sie können nur mit dem >>= aus der Monadeninstanz arbeiten, welches diesen Polymorphismus nicht hat. Da die `do`-Schreibweise doch sehr übersichtlich ist und deswegen hier oft eingesetzt wird, verzichten wir auch auf diese Möglichkeit.

Die angesprochene Umwandlung der einzelnen Konstrukte in einen `Transformer`, auf die die Definition von >>= zurückgreifen wird, ist simpel:

```
> mkTransformer :: DDAction a -> DDAction a
> mkTransformer m = case m of
>   Reader r       -> Transformer (\st -> (r st, []))
>   Appender (a,w) -> Transformer (const (a, [w]))
>   Ignorer a      -> Transformer (const (a, []))
>   transformer    -> transformer
```

Bevor wir die Monadeninstanz von `DDAction` deklarieren, wird noch eine Funktion gebraucht, die eine Liste von Kontextdefinitionen an den Zustand anhängt. Sie muß dabei die Aufteilung in dessen drei Komponenten beachten.

```
> appendDStateElems :: DState ->  [DStateElem] -> DState
> appendDStateElems st@(DState { dNameEnv = n
>                              , dTempNameEnv = t
>                              , dConEnv = c }) ns
>   = st {dNameEnv     = n ++ names,
>         dTempNameEnv = t ++ tempnames,
>         dConEnv      = c ++ con's}
>   where
>   (names, tempnames, con's) = foldr distr ([],[],[]) ns
>   distr (DNameDefElem n)     (a,b,c) = (n:a, b,   c)
>   distr (DTempNameDefElem n) (a,b,c) = (a,   n:b, c)
>   distr (DConDefElem n)      (a,b,c) = (a,   b,   n:c)
```

`monadInfo` läßt eine Monade auf einem Zustand „laufen“ und gibt neben ihrem Ergebnis und den anzuhängenden `DStateElems` auch den neuen Zustand zurück:

```
> monadInfo m st = let (a,app) = m' st in (a,app,appendDStateElems st app)
```

```
>   where m' = (deCons.mkTransformer) m

> deCons (Transformer t) = t
> deCons _               = undefined
```

Nun sind wir bereit, `DDAction` als Instanz der Monadenklasse zu schreiben. Auch wird es ein Mitglied der `Functor`-Klasse; ein passendes `fmap`, welches den Zustand in der Monade verändert, ist schon als `liftM` im Standard-Haskell-Modul `Monad` zu finden.

```
> instance Monad DDAction where
>   m >>= f = Transformer
>     (\st -> let (a,app1,st1) = monadInfo m st
>                 (b,app2) = f' a st1
>                 in (b, app1++app2))
>     where f' = deCons.mkTransformer.f

>   return = Ignorer

> instance Functor DDAction where
>   fmap = liftM
```

Nützlich ist eine Funktion, die eine Monade auf einen Zustand anwendet und nur das Ergebnis zurückgibt, den Folgezustand also verwirft (der Name geht auf [Jon95] zurück):

```
> m `startingWith` st
>   = (fst . deCons (mkTransformer m)) st
```

Ebenso sind die folgenden Funktionen gelegentlich von Nutzen. `peek` ist eine monadische Funktion, deren Wert der Zustand ist. `convertFuncMonad` und deren Verallgemeinerungen packen eine Funktion, deren Endergebnis monadisch ist, ganz in die Monade ein.

```
> peek :: DDAction DState
> peek = Reader id

> convertFuncMonad :: (a -> DDAction b) -> (DDAction (a -> b))
> convertFuncMonad2 :: (a -> b -> DDAction c)
>         -> (DDAction (a -> b -> c))
> convertFuncMonad3 :: (a -> b -> c -> DDAction d)
>         -> (DDAction (a -> b -> c -> d))

> convertFuncMonad f = do st <- peek
>                         return ((`startingWith` st) . f)

> convertFuncMonad2 = convertFuncMonadAux convertFuncMonad
> convertFuncMonad3 = convertFuncMonadAux convertFuncMonad2

> convertFuncMonadAux = ((.) convertFuncMonad) . (.)
```

7.1.4 Werkzeuge

Grundlegende Monadenoperationen

Von den Hilfsmitteln zum Umgang mit Kontexten ist zuerst, um beim vorigen Thema zu bleiben, der Zugriff auf die Kontextdefinitionen zu erwähnen.

`lookupNameDef` schlägt einen Kontext im Zustand der Monade nach. Es ist bequem, diese Operation auch für Jecs zuzulassen, indem dort jede einzelne Komponente nachgeschlagen wird. Damit können Kontexte der Form S `where` $T = (N_1, \ldots, N_n)$ sinnvoll benutzt werden; diese entstehen z.B. bei der Schnitt-/Vereinigungsbildung zweier `where`-Ausdrücke in der $\overset{\forall}{\rightarrow}$-Analyse.

```
> lookupNameDef (DName n) = Reader (\env ->
>     lookupError ("Demand.lookupNameDef:\
>                  \ DName "++n++" nicht definiert!")
>       n
>       (dNameEnv env))

> lookupNameDef (DConstraints cs) =  Reader (\env ->
>     lookupError ("Demand.lookupNameDef:\
>                  \ DConstraints "++show cs++" nicht definiert!")
>       cs
>       (dConEnv env))

> lookupNameDef (DJec a ns) = fmap (DJec a) $ mapM lookupNameDef ns

> lookupNameDef x = error ("Demand.lookupNameDef:\
>   \ 1. Argument "++show x++" ist weder Name noch Jec\
>   \ noch Constraints!")
```

Die Funktion `lookupError` schlägt dabei ihr zweites Argument im dritten nach und gibt bei Mißerfolg das erste Argument als Fehlermeldung aus.

Um einen oder eine Liste von neuen Kontexten in die Definitionen aufzunehmen, verwendet man die folgenden Funktionen. Ihr Ergebnistyp deutet an, daß sie keinen sinnvollen Wert berechnen; sie verändern nur den Monadenzustand:

```
> pokeDef :: (Name, DemandTerm) -> DDAction ()
> pokeDef d@(name, def)
>   =    lookupNameDefMaybe (DName name)
>    >>= maybe (Appender ((), DNameDefElem d))
>              ((const . Ignorer) ())

> pokeDefs :: [(Name, DemandTerm)] -> DDAction ()
> pokeDefs = sequence_ . map pokeDef
```

Wenn ein Name bereits definiert ist, so wird er nicht noch einmal hinzugefügt. Dafür ist `lookupNameDefMaybe` analog zu `lookupNameDef` definiert, nur gibt es einen Wert vom Typ `Maybe DemandTerm` zurück und verzichtet so auf Programmabbrüche.

Ähnliche Funktionen sind für die `dTempNameEnv`- und `dConEnv`-Komponenten der Kontextdefinitionen definiert.

Manchmal ist es angebracht, die Dualität von Namen und Constraints kurzzeitig aufzuheben und einen Vergleichsmöglichkeit unter einem einzigen Datentypen zu haben. Dazu sind die folgenden Funktionen definiert:

```
> constraints2Name :: [Constraint] -> Name
> constraints2Name = show

> getName :: DemandTerm -> Maybe Name
> getName (DName n)         = Just n
> getName (DConstraints cs) = Just (constraints2Name cs)
> getName _                 = Nothing
```

Konstruktion von Kontexten

Die nächsten Definitionen behandeln die Erzeugung von Kontexten. Ein leerer Kontext ist $\langle\rangle$:

```
> emptyDemand :: DemandTerm
> emptyDemand = DUnion emptySet
```

Jecs lassen sich ohne Angabe der Stelligkeit bilden; einstellige Jecs werden dabei nie mit dem Jec-Konstruktor versehen:

```
> mkDJec ds = let l = length ds in
>   case l of 1 -> head ds
>             _ -> DJec l ds
```

Vereinigungen und Schnitte bildet man auf folgende Weise leicht mit der Angabe von Listen statt von Mengen. Diese Funktionen werden durchgehend an Stelle der `DemandTerm`-Konstruktoren `DUnion` und `DIntersection` benutzt. Somit sind mehrere Invarianten, die überflüssige Berechnungen vermeiden helfen, auf einfache Weise garantiert:

- Direkte Teilkontexte von Schnitten sind niemals Schnitte und direkte Teile von Vereinigungen nie Vereinigungen. Durch die Verwendung von `flatten` wird nämlich eine solche redundante Hierarchiestufe abgebaut. Dies verspricht eine bessere Informationsverteilung in den Kontexten.
- In Schnitten kommt nie `Top` als direkter Teilkontext vor, da er herausgefiltert wird. Bei Vereinigungen entspricht dem die leere Vereinigung `DUnion emptySet`: auch diese wird, automatisch durch `flatten`, übergangen. Dies dient der Vermeidung von Redundanz und damit überflüssigen Auswertungen.
- Einelementige Vereinigungen und Schnitte kommen nicht vor; deren Teile stehen immer für sich. Auch so wird die Bildung überflüssiger Information schon im Ansatz verhindert.

```
> mkDUnion, mkDIntersection :: [DemandTerm] -> DemandTerm
> mkDUnion
>   = build DUnion        . list2Set . flatten isUnion
> mkDIntersection
>   = build DIntersection . list2Set . flatten isIntersection
>   . filter (/= (DName "Top"))

> flatten _ []     = []
> flatten pred (h:t) | pred h    = demParts h ++ rest
>                    | otherwise = h:rest
```

```
>   where rest = flatten pred t

> build constr s = case (setSize s) of 1 -> fst (popSet s)
>                                      _ -> constr s
```

Teilkontexte und rekursive Funktionen

Oben taucht die Funktion `demParts` auf. Sie gibt eine Liste der direkten Teilkontexte ihres Arguments zurück und ist weiter unten definiert.

Das führt uns zu Funktionen, die auf den Teilen eines beliebigen Kontextes operieren, möglicherweise sogar rekursiv. Solche Anwendungen sind vielfach vorhanden. Nur als Beispiele seien genannt: der Test auf Konventionalität, eine Substitution von Kontextvariablen in einem Gesamtkontext, Projektionen oder Komplemente.

Die Anzahl der Konstruktoren von `DemandTerm` ist aber recht hoch, so daß jede auf Kontexten rekursive Funktion ohne weitere Hilfsmittel viele Zeilen auf das Pattern Matching der einzelnen Konstruktoren verwenden müßte. Viele Zeilen, von denen die meisten eine im Prinzip gleiche Funktion darstellen und sich nur durch die verschiedene Syntax der Konstruktoren unterscheiden. Das ist redundant und fehleranfällig.

Wir wollen also zunächst einmal eine Funktion haben, die zu einem beliebigen Kontext die Liste seiner direkten Teilkontexte liefert. Als Vorbereitung dazu definieren wir eine Unterscheidung zwischen der *Art* eines Kontextes und seinen *Bestandteilen*:

```
> data DemandType
>   = BOT | FUN | NAME Name | VAR Name | CONSTR Tag Arity
>   | JEC Arity | UNION | INTER | WHERE | FORALL | EXISTS
>   | CONSTRAINTS [Constraint]
>   deriving (Show, Eq, Ord)
```

steht für die Art eines Kontextes. Die Teile haben natürlicherweise den Typ `[DemandTerm]`. Insgesamt wird so eine *uniforme* Darstellung von Kontexten ermöglicht.

```
> type UniList = [DemandTerm]
> type UniDemand = (DemandType, UniList)
```

Die Konversion von `DemandTerm` zu den beiden Komponenten von `UniDemand` ist einfach zu definieren:

```
> demType :: DemandTerm -> DemandType
> demType = fst . dt2Uni

> demParts :: DemandTerm -> [DemandTerm]
> demParts = snd . dt2Uni

> dt2Uni d = case d of
>   DBot                  ->  (BOT,        [])
>   DFun                  ->  (FUN,        [])
>   d@(DName name)        ->  (NAME name,  [])
>   u@(DUnion dset)       ->  (UNION,      set2List dset)
>   w@(s `DWhere` (t,n))  ->  (WHERE,      [s,t,n])
```

(die restlichen Zeilen der Definition von `dt2Uni` sehen analog aus). Damit können vielbenutzte Prädikate definiert werden, die die Art eines Kontexes überprüfen:

```
> isBot, isFun, isName, {-isVar,-} isConstr, isJec,
>   isUnion, isIntersection, isWhere, isForall, isExists
>   :: DemandTerm -> Bool
> isBot = (== BOT) . demType
> isFun = (== FUN) . demType
> isName = (\t -> case t of (NAME _) -> True; _ -> False) . demType
> isUnion = (== UNION) . demType

(...)

> isNaming = isName .||. isConstraints
```

`isVar` ist nur deshalb auskommentiert, weil dies der schon woanders deklarierte Name einer Methode einer Klasse `Term` ([Sch99a]) ist. Die Funktion wird jedoch auf `DemandTerm` so definiert, wie es hier gebraucht wird.

`.||.` ist eine in einem anderen Modul definierte Funktion, die zwei Prädikate auf ihren Ergebnissen mit `or` verknüpft, und ist [Man99] entnommen.

`isNaming` ist für den Test nützlich, ob es sich bei dem Argument um einen Kontext mit eigener Definition handelt. Die folgende modifizierte Version von `demParts` macht davon Gebrauch.

Beim Zugriff auf Teilkontexte ist es nämlich häufig erforderlich, die syntaktische Struktur auch über Kontextdefinitionen hinweg zu durchlaufen. Für diese Fälle sieht `demPartsM` im Gegensatz zu `demParts` die Definition eines Namens als *Teil* des Kontextes an. Um an diese zu gelangen, muß die Zustandsmonade benutzt werden; daher gründet sich auch das `M` im Namen.

```
> demPartsM :: DemandTerm -> DDAction UniList
> demPartsM d | isNaming d  = fmap (:[]) (lookupNameDef d)
>             | otherwise = Ignorer (demParts d)
```

Jetzt sind wir soweit, eine Schablone für rekursiv durch Kontexte gehende Funktionen zu konstruieren.

Das erste Argument des unten definierten `recGen` ist eine Funktion `f`, welche einen Kontext sowie eine Liste der Teilergebnisse (nämlich der Resultate auf den Teilen des Kontextes) erwartet. Der Ergebnistyp von `f` ist dabei beliebig (aber stimmt natürlich mit dem Typ der Teilergebnisse überein). Zweites Argument ist der Kontext, der bearbeitet werden soll.

`recGen` geht über eine reine Rekursion über Kontextteile hinaus: Mittels der Funktion `demPartsM` wird die Rekursion über Namensdefinitionen fortgesetzt. Namen, deren Berechnung ihres Funktionswertes schon in Angriff genommen wurde, werden gemerkt und solche Namen, wenn sie in der Rekursion erneut vorkommen, so betrachtet, als hätten sie eine leere Definition. Dieses Vorgehen vermeidet unendliche Berechnungen und stellt gleichzeitig eine natürliche Lösung für viele Zwecke dar.

```
> recGen :: (DemandTerm -> [a] -> a) -> DemandTerm -> DDAction a
> recGen f d = recGen_ns [] f d

> recGen_ns ns f d
>   =  do parts <- partsM
>         partResults <- mapM (recGen_ns new_ns f) parts
>         return (f d partResults)
```

```
>   where
>     name_schon_besucht        = (ns, return [])
>     name_noch_nicht_besucht n = (n:ns, demPartsM d)
>     normalfall                = (ns, demPartsM d)
>     treatName n = if (n `elem` ns)
>       then name_schon_besucht
>       else name_noch_nicht_besucht n
>     (new_ns, partsM) = case (getName d) of
>       Just n  -> treatName n
>       Nothing -> normalfall
```

Der Test auf Konventionalität eines Kontextes beispielsweise wird damit ein Dreizeiler (vier mit Typ):

```
> isConventional :: DemandTerm -> DDAction Bool
> isConventional = recGen f
>   where f d | (isForall .||. isExists) d = const False
>         f _                              = and
```

Die Gleichförmigkeit der Funktion auf den verschiedenen `DemandTerm`-Konstrukten ist damit sofort sichtbar.

Ein anderes Beispiel für die Anwendung von `recGen` ist die Bestimmung der ungebundenen Variablen eines Kontextes inklusive Mehrfachvorkommen:

```
> demandVarNamesMulti :: DemandTerm -> [Name]
> demandVarNamesMulti = (`startingWith` undefined) . recGen f
>   where f (DVar v)       = const [v]
>         f d | isNaming d = const []
>         f (DWhere _ _)   = \[ss,ts,_] ->
>                            filter (not . (`elem` ts)) ss
>         f d              = concat
```

Diese Funktion muß keine Namensdefinitionen nachschlagen. Sie muß also auch nicht monadischen Typs sein. Man kann also die durch `recGen` entstehende Monade auf einen beliebigen, da nicht gebrauchten Zustand anwenden und nur das Endergebnis entnehmen. Dieser Zustand kann ebensogut auch undefiniert sein, `undefined` — was seine fehlende Verwendung ausdrückt und die verzögerte Auswertung von Haskell ausnutzt.

Damit auch wirklich kein Name nachgeschlagen wird, bedarf es allerdings hier der expliziten Angabe, Namensdefinitionen zu ignorieren: daher die Zeile `f d | isNaming d = const []`.

Eine rekursiv über Teilkontexten arbeitende Funktion f kann auch selbst wieder einen Kontext als Ergebnis liefern. In diesem Fall findet eine Funktion `buildDemand` Anwendung, die einen Kontext C (z.B. das Argument von f) und eine Liste cs von Teilkontexten (z.B. den Teilresultaten) zu einem Ergebniskontext kombinieren soll, das der Art von C entspricht und alle Teile aus cs enthält. Dies gelingt mit Hilfe der Umkehrung von `dt2Uni`.

```
> buildDemand :: DemandTerm -> UniList -> DemandTerm
> buildDemand = curry uni2dt . demType

> uni2dt u = case u of
>   (BOT, _)            -> DBot
```

```
>   (FUN, _)            -> DFun
>   (NAME n, _)         -> DName n
>   (UNION, u)          -> mkDUnion u
>   (WHERE, [s,t,n])    -> s `DWhere` (t,n)
```

(Die anderen Zeilen lauten analog.) `buildDemand` kann nun vom ersten Argument von `recGen` benutzt werden.

Gelegentlich wird eine Variante von `recGen` gebraucht, die eine monadische Funktion anwenden kann und dabei die Monaden der Teilergebnisse hintereinander schaltet. Dies kann mit der Funktion `join` aus der Klasse `Monad` erreicht werden:

```
> recGenM :: (DemandTerm -> [DDAction a] -> DDAction a)
>     -> DemandTerm -> DDAction a
> recGenM f = join . recGen f
```

7.1.5 Inklusion, Konsistenz und Komplemente

Die Analysen höherer Ordnung machen Gebrauch von Tests auf $C_1 \leq C_2$ und $\gamma(C_1) = \emptyset$ für $C_1, C_2 \in \Lambda^C$ und von der Bildung von C^c für Kontexte C ohne `where`. Die Implementierung dieser Funktionen nimmt zuviel Platz ein, um hier wiedergegeben zu werden. Wir beschränken uns deshalb darauf, die Namen der Hauptfunktionen mit ihren Typen anzugeben und kurz zu beschreiben.

Alle Funktionen dieses Abschnitts außer der nachfolgenden Hilfsfunktion stützen sich auf theoretische Methoden aus [SSS97b].

Abschätzungen von `where`-Kontexten

Die verwendeten Kalküle aus [SSS97b] können nicht mit `where`-Kontexten umgehen. Jene sind aber esssentiell für die Rekursion in der $\overset{\forall}{\rightarrow}$-Analyse. Daher benutzen wir eine kleine und eine große Approximation von `where`-Ausdrücken (klein bzw. groß bezüglich der Konkretisierung) durch Kontexte, die ohne `where` auskommen. Diese Approximierung hat auch zur Folge, daß die Algorithmen dieses Abschnitts ein Ergebnis vom Typ `Maybe Bool` liefern, also eventuell keine Information herausfinden.

Inklusion

Der Test $C_1 \leq C_2$ ist genaugenommen als Test auf $\eta(C_1) \subseteq \eta(C_2)$ implementiert. Die beiden Formeln sind für nicht-konventionelle C_2 aber möglicherweise nicht gleichwertig (auch wenn noch keine C_1, C_2 bekannt sind, bei denen das der Fall ist). Es könnte daher im Prinzip vorkommen, daß die programmierte Inklusionsfunktion auf zwei Argumenten `Just False` ergibt, während die dem entsprechende theoretische Formel *wahr* wäre. Die Kontextanalyse ist aber so implementiert, daß sie auch in diesem Fall korrekt bleibt.

Die Implementierung nutzt die Gleichung $\eta(C_1) \subseteq \eta(C_2) \iff \eta(C_1 \cap C_2^c) = \emptyset$ aus, wie es auch schon in [SSS97b] benutzt wird. Sie greift also auf Konsistenz- und Komplementberechnungen zurück.

Die Hauptfunktion ist

```
> isIn :: DemandTerm -> DemandTerm -> DDAction (Maybe Bool)
```

Zusätzlich werden noch `isInT` und `isInF` desselben Typs definiert. Diese werden gebraucht, wenn es beim Aufruf nur auf die Information ankommt, ob der analoge Aufruf von `isIn` das Ergebnis

`Just True` oder nicht liefern würde; bzw. ob der Aufruf von `isIn` das Ergebnis `Just False` oder nicht liefern würde. In diesen Fällen muß somit nicht die gesamte Information, die `isIn` liefern würde, berechnet werden.

Konsistenz

Es wird ein Tableaukalkül aus [SSS97b] verwendet. Auch Kontexte höherer Ordnung können verarbeitet werden; dazu wird Lemma 4.36 benutzt.

Die Hauptfunktion heißt

```
> isConsistent :: DemandTerm -> DDAction (Maybe Bool)
```

und analog zu `isInT` und `isInF` sind zusätzlich `isConsistentT` und `isConsistentF` definiert.

Komplemente

Die Berechnung von Komplementkontexten findet auf Grundlage der Ausführungen in Abschnitt 4.8 statt. Die Vermutungen von dort werden hier übernommen.

Die Hauptfunktion ist

```
> complement :: DemandTerm -> DDAction DemandTerm
```

7.1.6 Projektionen

Für syntaktische Projektionen wird keine eigene Funktion definiert, da die syntaktischen Teile eines `DemandTerm` meistens als Liste schon vorliegen und an Ort und Stelle einfach bearbeitet werden können. Im folgenden geht es also noch um die großzügige bzw. kleinliche Projektion.

Sie werden in etwas vereinfachter Form definiert. Statt eines Kontextmusters benutzen wir als Parameter eine Liste von Zahlen, die die zurückzugebenden Konstruktorkomponenten angeben. Auch implementieren wir keine Projektion auf `where`-Kontexten. Für alle der später gemachten Beispielanalysen genügt nämlich die hier definierte Form, und Erkenntnisse über die Analysen können dort auch so schon gewonnen werden.

Projektionen werden in verschiedenen Zusammenhängen benutzt. Damit die Kontextanalyse an der jeweiligen Stelle immer korrekt bleibt, muß auf die Korrektheit bzw. Vollständigkeit der Projektionen selbst geachtet werden. Manchmal muß Korrektheit gegeben sein, d.h. es soll kein von der Konkretisierung her zu großer Kontext zurückgegeben werden, wie z.B. bei der Projektion auf dem in Verfahren 6.2 definierten N^{**}. In einem anderen Fall ist das genaue Gegenteil verlangt, also Vollständigkeit, nämlich die Projektion innerhalb der rekursiven $\overset{\forall}{\rightarrow}$-Analyse, die in der Formel (6.3) entsteht. Diese Projektion darf keine zu kleine Konkretisierung haben, denn sonst wird die damit benutzte Gesamtanalyse möglicherweise inkorrekt.

Da unsere Projektion nicht exakt arbeitet (auf Schnitten, `where`-Ausdrücken und wegen mancher Konsistenzabfragen), werden sie also einen Parameter benutzen, der angibt, ob korrekt oder vollständig gerechnet werden soll (englisch sound — korrekt, complete — vollständig):

```
> data SoundOrComplete = Sound | Complete
```

Es wird unterschieden zwischen Projektionen auf Jec-Komponenten und solchen bezüglich gewöhnlicher Konstruktoren. Die Jec-Projektion

```
> jecProj :: SoundOrComplete -> [Int] -> DemandTerm
>              -> DDAction DemandTerm
```

ist etwas einfacher, da sie z.B. nicht in Funktionskontexte weiter hineingreifen muß und auch nicht zwischen kleinlicher und großzügiger Projektion unterscheidet.

Ansonsten ist sie analog zu letzteren definiert, die wir nun vorstellen.

Die großzügige Projektion ist implementiert als

```
> proj :: SoundOrComplete -> Tag -> Arity -> [Int] -> DemandTerm
>           -> DDAction DemandTerm
```

`proj sc t a ps d` bedeutet eine Projektion von `d` auf die in `ps` angegebenen Komponenten des Konstruktors `Cons{t,a}`. Da die Funktion rekursiv ist und Monaden benutzt, ist sie über `recGenM` definiert.

```
> proj sc t a ps = recGenM f
>   where
>   f d parts
>     | (isBot .||. isFun .||. isJec) d = return emptyDemand
>     | isUnion d = fmap (buildDemand d) (sequence parts)
```

Für Funktionskontexte wird Lemma 4.36 benutzt.

```
>   f (DForall d1 d2) parts
>      = do d1consi <- isConsistent d1
>           case d1consi of
>             Just False -> projTop
>             Nothing   -> dontKnow
>             Just True ->
>               do botIn_d2 <- DBot `isIn` d2
>                  case botIn_d2 of
>                    Just True -> projTop
>                    Nothing -> dontKnow
>                    Just False -> return emptyDemand
```

Der Fall für `DExists` ist fast identisch zum `DForall`-Fall und wird deshalb ausgelassen.

Bei einem Konstruktorkontext wird von Lemma 4.55 Gebrauch gemacht und die Konsistenz nur für die Liste `restlist ps ds` der nicht zurückzugebenden Stellen untersucht. `restlist` und `sublist` sind weiter unten definiert.

```
>   f (DConstr ct ca ds) parts
>     | (t /= ct) || (a /= ca) = return emptyDemand
>     | ca >= last ps = do
>        restconsi <- isConsistent $ mkDJec $ restlist ps ds
>        return $ case restconsi of
>                   Just True  -> makeProj
>                   Just False -> emptyDemand
>                   Nothing -> case sc of
```

```
>                   Sound -> emptyDemand
>                   Complete -> makeProj
>     | otherwise = error "Projektion.proj: Der Konstruktor hat eine \
>                         \kleinere Stelligkeit, \
>                         \als die Projektion von ihm fordert!\n"
>     where makeProj = mkDJec $ sublist ps ds
```

Für Schnitte wird bei `sc == Complete` ausgenutzt, daß eine Definition der Form (4.11) zwar nicht immer korrekte, aber dafür vollständige Ergebnisse liefert.

```
>   f d parts
>     | isIntersection d = case sc of
>         Complete -> fmap (buildDemand d) (sequence parts)
>         Sound -> return emptyDemand
```

Bei einem Kontextnamen muß unterschieden werden, ob man ihn während der Rekursion schon behandelt hat (dann ist `parts` die leere Liste) oder nicht. In letzterem Fall muß ein Name für die Projektion definiert werden. Man beachte, daß dort entgegen der Definition 4.53 nicht dieser Name (`newname`), sondern dessen Definition `def` zurückgegeben wird. Somit kann z.B. die Projektion auf die zweite `Cons{2,2}`-Komponente von `Fin` wiederum `Fin` zurückgeben anstatt des neu definierten Namens `Proj[2]{2,2}(Fin)`.[1]

Der Name `Top` wird vorab gesondert behandelt, da das Ergebnis für ihn explizit angegeben werden kann, ohne neue Projektionsnamen einzuführen.

```
>   f d@(DName "Top")       _
>     = return $ mkDJec $ map (const (DName "Top")) ps
>   f d parts | isNaming d = case parts of
>         [defM] -> do def <- defM
>                      pokeDef (newname, def)
>                      return def
>         []     -> return (DName newname)
>     where newname = projName t a ps n
>           Just n  = getName d
>   f (DWhere s (t,n)) _ = dontKnow
>   projTop = f (DName "Top") undefined
>   dontKnow = case sc of Sound -> return emptyDemand
>                         Complete -> projTop
```

Hilfsfunktionen sind die Namensgebung sowie die Berechnung von Teillisten, welche durch Indizes gegeben sind.

```
> projName t a ps name = "Proj"++show ps++
>       "{"++show t++","++show a++"}("++name++")"

> sublist, restlist :: [Int] -> [a] -> [a]

> sublist ps xs = [ xs!!(i-1) | i <- ps ]
> restlist ps xs = sublist ps' xs
>   where ps' = [ i | i <- [1..length xs], i `notElem` ps ]
```

[1] Gemeint sind hier genaugenommen die entsprechenden Haskell-Ausdrücke vom Typ `DemandTerm`.

Die kleinliche Projektion heißt hier `pedanticProj`. Sie besteht aus einem Aufruf von `proj` mit vorgeschalteter Inklusionsprüfung. Bei Mißerfolg dieses Tests gibt sie `Nothing` (monadisch) zurück.

```
> pedanticProj :: SoundOrComplete -> Tag -> Arity -> [Int] -> DemandTerm
>       -> DDAction (Maybe DemandTerm)
> pedanticProj sc t a ps d
>   = do incl <- d `isInT` DConstr t a (replicate a (DName "Top"))
>        case incl of Just True -> fmap Just $ proj sc t a ps d
>                     _         -> return Nothing
```

7.2 Kernsprache

Der Datentyp für Kernsprachenausdrücke und die zugehörigen Funktionen (bis auf `isAbsApp`) wurden aus [Sch99a] übernommen.

```
> data LambdaCExpr
>   = LVar Name
>   | LAp LambdaCExpr LambdaCExpr
>   | LSuperC Name
>   | LConstr Tag Arity [LambdaCExpr]
>   | LCase LambdaCExpr [(Tag, [Name], LambdaCExpr)]
>   | LBot
>   | LJec Arity [LambdaCExpr]
>   | LPrim Name
>   deriving (Eq,Ord)
```

Die Namen der Konstruktoren sind selbsterklärend. Die Parameter von `LCase` entsprechen genau den Argumenten, die nach Abbildung 3.1 ein `case` haben muß. Die Variablen der Alternativen sind dabei gleich als `Name` codiert.

Ein Prädikat, das auf abstrakten Applikationen `True` ergibt, ist `isAbsApp`. Der Einfachheit halber soll es auch auf Variablen gelten; diese beiden Informationen werden nämlich immer zusammen abgefragt.

```
> isAbsApp f = case last (collectSpine f) of
>      LVar n -> True
>      _      -> False
```

Hierbei wird mit `collectSpine` auf das *Rückgrat* eines `LambdaCExpr`-Ausdrucks zugegriffen, also auf die aufgelöste Form einer Folge von Anwendungen. Diese Funktionalität wird auch z.B. bei der δ-Reduktion benötigt. Sie wird von der Klasse `Spine` bereitgestellt, von der `LambdaCExpr` eine Instanz ist:

```
> instance Spine LambdaCExpr  where
>   collectSpine l
>     = case l of
>         LAp e1 e2 -> e2:(collectSpine e1)
>         _ -> [l]
```

```
>   uncollectSpine l
>     = case l of
>         [x] -> x
>         x1:(x2:xs) -> LAp (uncollectSpine (x2:xs)) x1
```

Ein Programm ist codiert als eine Liste von Tripeln, die die Superkombinatoren, deren formalen Parameter und Definition enthalten:

```
> type TableauProgram = [(Name, [Name], LambdaCExpr)]
```

7.3 Tableaus

Die Darstellung der Tableaus und darauf operierender Funktionen übernehmen wir von [Sch99a], wo seinerseits auf [Man99] zurückgegriffen wurde. Matthias Mann hat nämlich in seiner Diplomarbeit ein Konzept bezüglich Tableaus eingeführt, das einerseits nach außen sehr modular ist, so daß es auch sehr gut für die Kontextanalyse eingesetzt werden kann. Zum anderen ist auch die interne Modularisierung sehr stark. Der klare Aufbau überzeugt vor allem in den folgenden Punkten:

- Es gibt eine klare Trennung zwischen den *Regelanwendungen*, also den theoretisch möglichen Expansionen, und der *Heuristik*, also der Auswahl einer Expansion aus mehreren möglichen.
- Der für die Markierung der Tableauknoten benutzte Datentyp ist vollkommen frei wählbar.

Das Konzept wird in [Man99] ausführlich besprochen; wir gehen deshalb hier nur auf die wichtigsten Punkte ein.

7.3.1 Arbeitsgänge

Die Anwendung eines Tableaukalküls wird von der Anfrage bis zur Lösung in verschiedene Phasen gegliedert:

1. **Expansion:** Von der Wurzel ausgehend wird durch fortgesetzte Anwendung aller passenden Expansionsregeln eine Menge von Tableaus erzeugt.
2. **Metaregeln:** Die Schleifenregeln werden, soweit möglich, auf allen Knoten aller Tableaus angewandt.
3. **Heuristik:** An jedem Knoten wird jeweils eine der anwendbaren Regeln ausgewählt. Damit bleibt ein einzelnes Tableau übrig.
4. **Beschneidung:** Gegebenenfalls muß ein unendliches oder zu großes Tableau ab einem bestimmten Niveau abgeschnitten werden.
5. **Lösung:** Das fertige Tableau wird als Lösung[2] (hier: Kontext) interpretiert.

Die Verarbeitung potentiell unendlicher Tableaus macht keine Schwierigkeiten, da Haskell ja verzögert auswertet. Ganz im Gegenteil werden die Tableaus der ersten beiden Schritte nur so weit ausgewertet, wie es die Heuristik erfordert. Es sei hiermit betont, daß die ersten beiden Schritten gleichwohl alle nur denkbaren geschlossenen und nicht geschlossenen Tableaus produzieren.

Auf die einzelnen Phasen und die Datenstrukturen wird im folgenden kurz eingegangen. Dabei stellen wir fest, was in [Man99] programmiert wurde und an welchen Stellen die Kontextanalyse noch etwas tun muß.

[2] Der Begriff der *Lösung* eines Tableaus wird hier und im folgenden als „der von einem Tableau repräsentierte Kontext“ gedeutet und ist nicht mit dem theoretisch definierten Begriff zu verwechseln.

Expansionsregeln und Mengen von Tableaus

Das vielleicht grundlegendste der Konzepte im Zusammenhang mit Tableaus sind die Expansionsregeln. Diese bergen in sich sozusagen die Struktur eines Tableaus im Kleinen, nämlich die Verzweigung. Aus einem Knoten entstehen null oder mehr Nachfolger. Diese variable Anzahl von Elementen wird zusammen mit dem Typ der Verzweigung dargestellt im Datentyp

```
> data Various a = VNothing
>                | VJust a
>                | VOr [a]
>                | VAnd [a]
>                deriving (Show,Eq)
```

der auch auf naheliegende Weise eine Instanz der `Functor`-Klasse ist.

Der für eine konjunktive Verknüpfung der Untertableaus stehende Konstruktor `VAnd` wird für die Kontextanalyse nicht benötigt, da dort jede Verzweigung eine Oder-Beziehung bedeutet.

Eine Expansionsregel hat damit den Datentyp

```
> type ExpansionRule a = (a -> Various a)
```

Regeln dieses Typs sind also für die Kontextanalyse anzugeben. Über die konkrete Ausgestaltung s.u.

An einem Knoten sind nun wiederum null oder mehrere Regeln anwendbar. Diese werden in einer Liste dargestellt, so daß sich die folgende Struktur für Mengen von Tableaus ergibt. Das `a` steht dabei wie oben schon für den Knotentyp.

```
> data TableauSet a = TSBranch [Various (TableauSet a)] a
>                   | TSLeaf a
>                   deriving (Show,Eq)
```

Ein Verzweigungsknoten hat also eine Liste von Nachfolgern, wobei jeder Nachfolger eine durch eine Expansionsregel gebildete Verzweigung darstellt. Diese Verzweigung verweist dann auf die eigentlichen Nachfolger-Tableaumengen.

Auch auf `TableauSet` ist eine Instanz von `fmap` definiert.

Die Anwendung der Regeln, also die Expansion der Tableaumengen erfolgt durch die Funktion `expandTableau`, von der wir hier den Typ angeben:

```
> expandTableau :: (a -> Bool) -> [ExpansionRule a] ->
>                  TableauSet a -> TableauSet a
```

Der erste Parameter ist ein Prädikat, das angibt, an welchen Knoten nicht weiterexpandiert werden soll. In der Kontextanalyse sind das diejenigen, die Definition 5.2 erwähnt (ausgenommen den *loop!*-Knoten, die erst nach der Expansionsphase entstehen). Gestartet wird der Vorgang auf einem Wurzelknoten `e`, indem `expandTableau` auf der initialen Tableaumenge `TSBranch [] e` aufgerufen wird.

Metaregeln

Bei der Expansion wurden die Schleifenregeln außen vor gelassen. Sie arbeiten im Gegensatz zu den Expansionsregeln nämlich nicht *lokal*, d.h. auf einem einzelnen Knoten, sondern benötigen Information über zwei Knoten und den Pfad von der Wurzel herunter. Deswegen werden die Schleifen- auch Metaregeln genannt.

Die Schleifenbildung generiert alle möglichen Schleifen. Der Aufruf geschieht durch die Funktion

```
> loopTableau :: PathRule rule => (a -> a -> Maybe a) ->
>                [rule a] -> [rule a] ->
>                TableauSet a -> TableauSet a
```

Deren erstes Argument gibt den Knoten an, der bei einer Schleifenbildung entsteht. Er ist abhängig von den beiden Knoten, die miteinander die Schleife bilden, und kann — wenn sich keine Schleife ergibt — auch `Nothing` sein.

Die Parameter des Typs `PathRule rule => [rule a]` geben Bedingungen für den Pfad von der Wurzel zum oberen Schleifenknoten, bzw. zwischen den beiden Schleifenknoten an. Auf deren Darstellung wird hier nicht näher eingegangen, da alle hier benutzten Pfadregeln schon in [Sch99a] definiert wurden.

Heuristik, Beschneidung und Lösung

Für die übrigen Phasen einer Analyse werden von [Man99] keine besonderen Konstrukte bereitgestellt, außer für das Abschneiden eines Tableaus, was jedoch sehr einfach als `takeTableau` analog zum `take` auf Listen definiert wurde.

Die Heuristik und die Lösungsbildung werden weiter unten direkt programmiert.

7.3.2 Ein einzelnes Tableau

Was bis jetzt noch fehlt von den Tableau-Strukturen, ist die Darstellung eines einzelnen Tableaus. Sein Typ ist wie `TableauSet` definiert, nur daß jetzt alle Nachfolger durch eine Regel erzeugt werden, also jeweils genau ein `Various`-Konstrukt sind:

```
> data Tableau a = TBranch (Various (Tableau a)) a
>                | TLeaf a
```

7.4 Überlegungen zur Tableau-Organisation

Bevor wir die Konstrukte aus dem vorigen Abschnitt auf konkreten Tableaus benutzen, stehen einige Gedanken zum Aufbau der Tableaus an. Es geht zum einen um die Rolle der Analysen höherer Ordnung; zum anderen geht es um den Knoteninhalt.

7.4.1 Einbinden der Analysen höherer Ordnung in den Tableaukalkül

Bei den Analysen aus Kapitel 6 handelt es sich um Berechnungen, die über den Tableaukalkül hinausgehen. Anders formuliert, haben dort Teile der Berechnungen nichts zu tun mit Tableaus und Regeln. Jedoch wird im allgemeinen auch auf Anfragen an den Kalkül zurückgegriffen.

Diese Teilabhängigkeit von den Tableaurechnungen wirft die Frage auf, wie Kalkül und Analysen zueinander in Beziehung gebracht werden sollen. So kann z.B. die Anfrage $f \in C \overset{\forall}{\rightarrow} D$ innerhalb eines Tableaus auftauchen, beispielsweise weil f durch eine Regel wie die no-Reduktion aus einer anderen Anfrage entstanden ist.

Angenommen, das aktuelle Tableau soll auf eine bestimmte Tiefe beschränkt werden. Nehmen wir weiter an, daß die Anfrage $f \in C \overset{\forall}{\to} D$ nun von der Analyse bearbeitet wird. Diese benutzt ihrerseits die Anfrage $f\ y \in D$, für die ein neues Tableau angelegt wird. Wie tief soll nun dieses Tableau ausgewertet werden? Besser nicht genauso tief wie das erste Tableau, denn das könnte eventuell zu nichtterminierenden Berechnungen führen. Eine konstante Tiefe wäre dagegen eine unnötige Beschränkung.

Eine natürliche Lösung ist es, das neue Tableau in das alte Tableau einzufügen. Wenn man sich die Anfragen $f \in C \overset{\forall}{\to} D$ und $f\ y \in D$ anschaut, erkennt man auch, wie ähnlich diese sind, und das legt nahe, eine eigene Expansionsregel dafür einzuführen.

Eine Tableauregel für die Analysen

So definieren wir eine Quasi-Regel

$$\frac{f \in C \overset{*}{\to} D}{f\ y \in D}, * \in \{\forall, \exists\}, y \text{ nicht in } f \text{ vorkommende Variable,} \qquad \text{(AddArg)}$$

die nur für das Einbinden der Analysen in die Tableaus gedacht ist. Natürlich darf sie nicht als theoretische Kalkülregel aufgefaßt werden, denn Eltern- und Kindknoten haben ja unterschiedliche Lösungen, und außerdem wird nicht zwischen $\overset{\forall}{\to}$- und $\overset{\exists}{\to}$-Kontexten unterschieden.

Vielmehr ist es nun Aufgabe der Lösungsfunktion, die einzelnen Analysen aufzurufen; also derjenigen Funktion, die für ein Tableau den repräsentierten Kontext ausgibt.

Eine solche Regel läßt ein Tableau theoretisch in zwei Teiltableaus zerfallen. Sie bildet eine Grenze, die bei der Schleifenbildung nicht überschritten werden darf. Dies ist an gegebener Stelle weiter unten zu beachten, ebenso, wie eben schon erwähnt, bei der Lösung des Tableaus. In einem Pfad unterhalb einer solchen Regelanwendung werden wir uns deshalb in allen Knoten Informationen darüber merken:

```
> type AddedVarDem = (Name, DemandTerm, DemandType)
> type AddedVarDems = [AddedVarDem]
```

Die erste Komponente der Tripel steht dabei für den Namen der Variablen y aus der obigen Regel. Der zweite Teil steht für den Kontext C, und der letzte bezeichnet die Art `FORALL` oder `EXISTS`. Die Informationen werden für den Fall mehrerer dieser Regelanwendungen in einer Liste aufgesammelt.

Modifizierte Schleifenerkennung

Wir wollen sicherstellen, daß die Schleifenbildung einerseits nie zwei Knoten betrifft, bei denen eine Anwendung der Regel (AddArg) auf dem Pfad zwischen ihnen liegt. Andererseits soll sie innerhalb jedes einzelnen, durch Anwendungen von (AddArg) voneinander getrennten Teiltableaus funktionieren.

Man könnte versuchen, diese Bedingungen als Pfadregel zu formulieren; das ist der in [Man99] vorgesehene Mechanismus für solche Vorbedingungen von Schleifenbildungen.

Genauer betrachtet, müßten sogar die Implementierungen aller in Abschnitt 5.1.2 aufgeführten Bedingungen umcodiert werden, die sich auf den Pfad von der Wurzel hinab beziehen (z.B., daß oberhalb der Schleifenknoten keine `case`-Verzweigung liegt). Denn das Tableau unterhalb einer (AddArg)-Regelanwendung soll ja wie ein eigenständiges Tableau behandelt werden. Somit müßten diese Pfadregeln nach dem Auftauchen einer (AddArg)-Regel alle neu initialisiert werden.

Zudem ist die von Matthias Mann implementierte Instanz des Pfadregeltyps nur für Fälle geeignet, in denen sich die Wahrheitswerte der Pfadbedingungen für Teilpfade höchstens einmal innerhalb eines Pfades ändern. Auch das müßte angepaßt werden.

Anstatt aber an vielen Stellen Änderungen durchzuführen, ist es einfacher, die Funktion `loopTableau` direkt zu ändern. Dies stellt sich als relativ simpel heraus, denn die Funktion durchläuft das Tableau von der Wurzel hinab. Bei diesem Durchlaufen werden an jedem Knoten die

verwendeten Pfadregeln angepaßt an die bis dahin schon gemachten Beobachtungen über den zurückgelegten Pfad. Es genügt somit, die initial benutzten Pfadregeln für den Pfad von der Wurzel herab sich in einem zusätzlichen Argument zu merken, bei jedem Knoten auf Anwendung der (AddArg)-Regel zu überprüfen (das geht über die `RuleID`, s.u.) und wenn dies gilt, dann statt mit den aktuellen Pfadregeln eben mit den gemerkten initialen Pfadregeln weiterzumachen.

Die modifizierte Version heißt `loopTableauHO` (da die Änderung nur wegen der Analysen höherer Ordnung gemacht wurde). Der Code ist fast identisch zum `loopTableau` aus [Man99], weswegen er hier nicht extra aufgeführt wird.

7.4.2 Die konjunktive Struktur der Knoten

Der Typ der Tableauknoten ist weitgehend aus [Sch99a] übernommen. Eine Eigenschaft war dort jedoch noch unberücksichtigt: Jeder Knoten stellt ja eine Konjunktion seiner Constraints dar. Das ist im Prinzip ganz analog zur disjunktiven Struktur der Verzweigungen im Tableau. Nun könnte man dafür den Konstruktor `VAnd` verwenden, wenn man diese Und-Verknüpfung ebenfalls als Tableau-Verzweigung darstellen könnte.

Dies würde eine Auftrennung der Knoten bedeuten und läßt sich leider nicht realisieren, da manche der Regeln auf alle Constraints des Knotens zugreifen müssen. Z.B. müssen die `case`-Regeln nicht nur das `case`-Constraint bearbeiten, sondern auch die betreffende Variable in den restlichen Constraints substituieren.

Vereinfachter Knotentyp

Nichtsdestotrotz beziehen sich die meisten Regeln nur auf ein Constraint und lassen die anderen ganz unberührt. Es würde genügen, diese Regeln auf einem verkleinerten Knotentyp zu definieren, der nur ein Constraint enthält. Da Constraints auch gestrichen werden können, müßte dazu auch die Möglichkeit von *keinem* Constraint im Knoten einbezogen werden.

Wir werden in der Implementierung diesem Ansatz nachgehen, weil er mehr Übersichtlichkeit verspricht. Das heißt: Die *Tableaus* müssen zwar auf Knoten mit mehreren Constraints operieren. Die meisten *Regeln* aber werden auf einem vereinfachten Knotentypen arbeiten, der null oder ein Constraint enthält. Diese Regeln können sinnvoll mit `VAnd` arbeiten. Sie werden dann für die Expansion gemeinsam zum allgemeineren Typ konvertiert.

Die Vereinigungsregel z.B. könnte man so als

$$\frac{s \in \langle C_1, \ldots, C_m \rangle, \sigma}{s \in C_1, \sigma | \cdots | s \in C_m, \sigma}$$

ohne die übrigen Constraints notieren. Deren Implementierung benutzt den Konstruktor `VAnd` für die einzelnen Ergebnisconstraints, anstatt sie zusammen in einer Liste unterzubringen.

Hyperregeln

Nun ist es so, daß auch solche Regeln, die auf den ganzen Knoten zugreifen müssen, nur von einem Constraint abhängen. Die Anwendbarkeit der `case`-Regeln z.B. hängt nur vom Vorhandensein eines Constraints mit einem `case` ab.

Es können aber auch mehrere dieser Constraints in einem Knoten vorkommen. Das würde bedeuten, daß die entsprechende Regel an verschiedenen Stellen desselben Knotens anwendbar wäre, mit jeweils unterschiedlichem Ergebnis. So kann eine einzige Regel an einem Knoten im Prinzip beliebig (endlich) oft anwendbar sein.

Das führt uns dazu, jede Expansionsregel als *Hyperregel* zu sehen, also als Regel, deren Anwendung nicht wie sonst nur eine Verzweigung des Tableaus erzeugt, sondern beliebig viele. Diese Struktur wird durch den Typ

```
> type HyperExpansionRule a = (a -> [Various a])
```

dargestellt.

Wir werden alle Regeln, die nicht auf dem oben erwähnten vereinfachten Knotentyp arbeiten, als normale Expansionsregeln so schreiben, daß sie nur das erste Element der Constraintliste des Knotens berücksichtigen. So ist es auch in [Sch99a] programmiert. Für die Expansionsphase werden sie aber durch eine Konvertierungsfunktion in eine Regel des Typs `HyperExpansionRule` umgewandelt, der durch den enthaltenen Listentyp eben alle möglichen Anwendungen der Regel beachtet.

Dafür ist nun die Funktion `expandTableau` nicht mehr adäquat; es wird statt dessen ein `hyperExpandTableau` benutzt, das die Funktionalität von `expandTableau` erweitert und alle durch den neuen Regeltyp entstehenden Knoten berücksichtigt.

7.5 Ausfüllen des `TableauSet`-Gerüsts

Nun soll das Gerüst der Tableaumengen-Maschinerie mit Kontextanalyse-spezifischem Code ausgefüllt werden.

7.5.1 Die verwendeten Typen für die Tableaus

Die Kontextanalyse benutzt ein Tableau des Typs

```
> type DemandTableau = Tableau DemandNode
```

mit einem passenden Typ `DemandNode` für die Knotenmarkierung. Er benutzt einen Hilfstyp, der polymorph über dem Typ der Constraints ist:

```
> data DemandNode' a
>     = DemandNode { ruleID :: RuleID
>                  , constraints :: a
>                  , ccvs :: [CCV]
>                  , sigma :: Substitution LambdaCExpr
>                  , addLabel :: Maybe AdditionalLabel
>                  , oVars :: [Name]
>                  , nextName :: Int
>                  , addedVarDems :: [AddedVarDem]
>                  }
>     deriving (Eq)
```

Die erste Komponente `ruleID` ist eine Kurzbeschreibung der Regel, die den Knoten hervorgebracht hat; der Typ `RuleID` wird weiter unten definiert. Die letzte Komponente `addedVarDems` wurde schon in Abschnitt 7.4.1 besprochen. Die anderen Felder sind aus [Sch99a] übernommen: Es sind Constraints, CCVs und eine Substitution; eine Marke `addLabel` als Repräsentation von *loop!* w und *no!*; sowie eine Liste `oVars` der freien Variablen der Wurzel und ein Hilfsfeld `nextName`, das für die Einführung neuer Variablen benutzt wird.

Es werden im weiteren immer wieder Funktionen im Zusammenhang mit Substitutionen benutzt werden. Diese stammen aus einem Modul aus [Sch99a], so daß wir hier nicht näher auf deren Funktionsweise eingehen.

Die Constraints sind nun nach den obigen Ausführungen normalerweise eine Liste, aber bei vielen Regeln, wie in Abschnitt 7.4.2 angekündigt, als `Maybe Constraint` dargestellt, was also nur null oder einen Constraint zuläßt.

```
> type DemandNode = DemandNode' [Constraint]
> type SimpleDemandNode = DemandNode' (Maybe Constraint)
```

Die Marken für *no!* oder *loop! w* sind im Typ `AdditionalLabel` festgelegt:

```
> data AdditionalLabel
>   = No
>   | Loop ([Name] -> DemandTerm) DemandNode
```

In `Loop f y` ist `y` der Knoten, mit dem die Schleife gebildet wird. `f` ist der von einer Variablenliste abhängige `where`-Ausdruck, der bei einer Schleife entsteht. Der Ausdruck fordert diesen Parameter, weil er in den Lösungen von verschiedenen Teiltableaus auftreten kann, siehe dazu Abschnitt 7.7.

Der Typ `RuleID` enthält die Kurzbezeichnungen aller Regeln nach einer Idee in [Man99]. Die Reihenfolge der Aufzählung ist so gewählt, daß sie später für Heuristik benutzt werden kann.

```
> data RuleID = Initial
>             | NoSolution
>             | RedundantTop
>             | RedundantBot
>             | SimpleVar
>             | ConsDecomp
>             | UnionDecomp
>             | JecDecomp
>             | Reuse
>             | JecInConstraint
>             | CaseRule
>             | ConsWithDVar
>             | BotWithDVar
>             | DissolveAp
>             | DeltaRed LambdaCExpr --Argument ist der Redex
>             | NameLookup
>             | DemConstraint
>             | AddArg
>  deriving (Eq, Ord, Show)
```

Die Regeln mit den Kürzeln `ConsWithDVar`, `BotWithDVar`, `DissolveAp` und `AddArg` werden in dieser Arbeit implementiert. Die Marke `Initial` bekommt die Wurzel. Alle anderen Werte beziehen sich auf Regeln, die bereits in [Sch99a] definiert sind.

Die Einführung der Regelkürzel in jedem Knoten dient nicht nur oberflächlich der Übersichtlichkeit: Viele der Pfadbedingungen, die für die Schleifenregeln überprüft werden müssen, können jetzt anhand der `ruleID` nachgeschlagen werden. Somit sind hier einige Felder nicht mehr nötig, die in [Sch99a] noch Komponenten des Knotentyps waren und Informationen lieferten etwa über das Vorhandensein einer Konstruktordekomposition im Pfad.

7.5.2 Die Tableauregeln für die Behandlung höherer Ordnung

Nun wird die Programmierung der drei Regeln aus Abschnitt 5.2 vorgestellt, sowie von der Regel (AddArg). Am Schluß des Abschnitts wird auf eine zweite Regel für `case`-Ausdrücke aus [Sch99a] eingegangen, die auch für Argumente höherer Ordnung zur Geltung kommt.

`dissolveAp`

Die Regel `dissolveAp` berührt nur einen Eltern- und zwei Kinderconstraints, also nicht den ganzen Knoten. Sie kann deswegen als `ExpansionRule SimpleDemandNode` geschrieben werden.

Die Regel ist so implementiert, daß bei einer Anwendung $f\ y$, wenn y eine Variable ist, die neue Kontextvariable zur besseren Übersicht ebenfalls y genannt wird und auf die Erzeugung des Constraints $y \in y$ verzichtet wird. Die Existenz von Constraints dieser Art wird automatisch angenommen. Dies wird in der `case`-Regel berücksichtigt: In [Sch99a] ist sie nämlich bereits so programmiert, daß Substitutionen von Λ-Variablen auch auf der Kontextseite vorgenommen werden.

```
> dissolveAp :: ExpansionRule SimpleDemandNode
> dissolveAp label@DemandNode
>   { constraints = Just (LAp f t :<-: d)
>                 , ccvs = ccvs
>                 , nextName = nextName }
>   | isAbsApp f
>   = listToVariousWith VAnd newlabels
>   where firstconstraint = (f :<-: DForall newvar d)
>         newlabels = map (\c -> label' { constraints = Just c }) $
>             firstconstraint : case t of
>                               LVar n -> []
>                               _      -> [t :<-: newvar]
>         label' = label { ruleID = DissolveAp, nextName = nextName + 1 }
>         newvar = DVar $ case t of
>             LVar n -> n
>             _      -> "_x_"++show nextName

> dissolveAp _ = VNothing
```

`listToVariousWith VAnd` bezieht sich auf eine in [Man99] definierte Funktion und bedeutet, daß die entstehenden Constraints konjunktiv miteinander verknüpft werden, also keine Tableauverzweigung nötig ist. Der Name für die neue Kontextvariable wird mit Hilfe von `nextName` gebildet, um nicht eine schon vorhandene Variable zu erzeugen.

`botWithDVar`

Die anderen beiden Regeln aus Abschnitt 5.2 beinhalten eine Substitution, die über alle Constraints läuft. Die Implementierung erfolgt deswegen nicht über dem vereinfachten Knotentypen.

```
> botWithDVar :: ExpansionRule DemandNode
> botWithDVar label@DemandNode
>                     { constraints = l@LBot :<-: DVar y : exs
>                     , ccvs = ccvs
>                     , sigma = sigma }
>   = VJust $ label' { constraints = exs'
>                        , ccvs = ccvs'
>                        , sigma = sigma' newSubst }
>   where label' = label { ruleID = BotWithDVar }
>         subst = S [y :->: DBot]
>         newSubst = S [y :->: l]
>         sigma' ss@(S [x]) = S $ x:(dS $ specializeSubst ss sigma)
>         exs'  = [ applySubst newSubst l :<-: applySubst subst d
```

```
>                  | l :<-: d <- exs ]
>         ccvs' = [ l `In` applySubst subst d | l `In` d <- ccvs ]

> botWithDVar _ = VNothing
```

Die auf alle CCVs und Constraints anzuwendende Substitution $y \mapsto$ `Bot` aus der Theorie ist hier als `subst` zu finden.

In [Sch99a] gibt es Regeln, die ebenfalls bei einem Constraint `bot` $\in y$ bzw. wie unten bei `c` $t_1 \ldots t_n \in y$, y Variable, Anwendung findet. Diese wurden mit in `botWithDVar` bzw. `consWithDVar` (s.u.) eingebaut. Sie sind jeweils in der Anwendung von `sigma'` und `newSubst` zu finden und führen Substitutionen auf der Λ-Seite von Constraints durch. Dieses Vorgehen hängt zusammen mit der zusätzlichen `case`-Regel (s.u.) und wurde vom Autor der bewußten Regeln noch nicht in der Korrektheit verifiziert. Wollte man diese Regeln deswegen ausschließen, so müßte man `newSubst` an die leere Substitution `S []` binden und `sigma' = id` setzen.

`consWithDVar`

Ganz ähnlich wie `botWithDVar` ist `consWithDVar` programmiert. Hier müssen jedoch auch neue Kontextvariablen erzeugt werden. Dazu wird wieder, mehrfach, `nextName` zu Hilfe genommen.

```
> consWithDVar :: ExpansionRule DemandNode
> consWithDVar label@DemandNode
>                       { constraints = l@(LConstr t a _) :<-: DVar y : exs
>                       , ccvs = ccvs
>                       , sigma = sigma
>                       , nextName = nextName }
>   = VJust $ label' { constraints = l :<-: newCons : exs'
>                         , ccvs = ccvs'
>                         , sigma = sigma' newSubst
>                         , nextName = nextName + a }
>   where label' = label { ruleID = ConsWithDVar }
>         newCons = DConstr t a getNewVarNames
>         getNewVarNames = map (DVar . ("_x_"++) . show) $ take a [nextName..]
>         sigma' ss@(S [x]) = S $ x:(dS $ specializeSubst ss sigma)
>         newSubst = S [y :->: l]
>         subst = S [y :->: newCons]
>         exs'  = [ applySubst newSubst l :<-: applySubst subst d
>                 | l :<-: d <- exs  ]
>         ccvs' = [ l `In` applySubst subst d | l `In` d <- ccvs ]

> consWithDVar _ = VNothing
```

`addArg`

Die Implementierung der in Abschnitt 7.4 vorgestellten Regel (AddArg) muß die Information `addedVarDems` aktualisieren. Wieder wird `nextName` benutzt, diesmal für eine Λ-Variable.

```
> addArg :: ExpansionRule DemandNode
> addArg label@DemandNode
>                       { constraints = (l :<-: d) : rest
>                       , addedVarDems = addedVarDems
```

```
>                     , nextName = nextName
>                     , oVars = oVars
>                     }
>   = case d of
>     (DForall d1 d2) -> result FORALL d1 d2
>     (DExists d1 d2) -> result EXISTS d1 d2
>     _               -> VNothing
>   where
>     result typ d1 d2
>       = VJust $ label { constraints = newConstraints
>                       , addedVarDems = (newVarName, d1, typ):addedVarDems
>                       , nextName = nextName + 1
>                       , oVars = constraints2VarNames newConstraints
>                       , ruleID = AddArg
>                               }
>       where newConstraints = (LAp l newVar :<-: d2):rest
>     newVarName = "x_"++show nextName
>     newVar = LVar newVarName
>     constraints2VarNames = concat . map (freeVars . expr)
```

Eine zweite case-Regel

An dieser Stelle sei hinzugefügt, daß aus [Sch99a] eine Regel übernommen wurde, die auch für Funktionen höherer Ordnung interessant ist, deren Korrektheit aber noch nicht vollständig bewiesen ist. Es handelt sich um die Implementierung einer zweiten case-Regel, die in der Theorie folgendes Aussehen hätte:

$$\frac{D[\mathtt{case}_A\ t\ e_1 \ldots e_{|A|}] \in C, R, \sigma}{D[\mathtt{case}_A\ x\ e_1 \ldots e_{|A|}] \in C, t \in x, x \in x, R, \sigma}, \ x \text{ neue } \Lambda\text{- und Kontextvariable}$$

Sie wird dann eingesetzt, wenn t weder eine Variable ist noch in SCWHNF steht, also keine andere Regel greift. Nützlich ist sie, wenn z.B. innerhalb eines case-Ausdrucks eine Variable als Argument höherer Ordnung auf einen anderen Ausdruck angewendet wird und dadurch das obige t für eine abstrakte Applikation steht.

7.5.3 Heuristik, Lösung und Aufruf der Phasen

Heuristik

Eine Heuristik kann nie endgültig sein in dem Sinne, daß sie für alle möglichen Anfragen gleichermaßen optimal wirkt. Denn einerseits ist nicht klar, was hier „optimal“ bedeutet: Etwa das kürzeste Tableau zu erzeugen, oder vielleicht die kürzeste oder wie auch immer lesbarste Lösung? Mehr Antworten sind denkbar. Andererseits ist ja die Heuristik nur eine lokale Entscheidungshilfe, die nicht die gesamte Weiterentwicklung des Tableaus vorhersagen kann.

Als grundsätzlich brauchbar hat sich jedoch eine Methode erwiesen, die wann immer möglich eine Regel auswählt, die ein Blatt produziert, und ansonsten eine bestimmte Rangfolge der Regeln benutzt.

Diese Rangfolge ist hauptsächlich diejenige, die durch die Reihenfolge der Konstruktoren des Typs RuleID vorgegeben ist. Dort stehen prinzipiell die Kürzel von solchen Regeln weiter oben (was eine größere Priorität bedeutet), die die Struktur eines Knotens vereinfachen, indem sie z.B. Constraints entfernen. Weiterhin wird dort beispielsweise eine δ-Reduktion gegenüber der Anwendung der AddArg-Regel bevorzugt, damit z.B. ein Constraint $\mathtt{const}\ x\ y \in \mathtt{Fin} \overset{\forall}{\rightarrow} \mathtt{Inf}$ auch ohne Analysen höherer Ordnung behandelt werden kann.

Die genaue Rangordnung wird durch `before` gegeben. `x 'before' y` ergibt `True`, wenn `x` gegenüber `y` bevorzugt wird. Die einzigen Ausnahmen gegenüber der durch die Ordnung von `RuleID` gegebenen Hierarchie sind die, daß die Anwendung eines Konstruktorausdrucks auf ein weiteres Argument eine relativ hohe Priorität bekommt, und daß die Reduktion von `bot` grundsätzlich als letztes passiert.

```
> before :: RuleID -> RuleID -> Bool

> DeltaRed x    `before` rid2
>   | isConstrApp x                  = SimpleVar < rid2
> DeltaRed LBot `before` rid2        = False
> rid1          `before` DeltaRed LBot = True
> rid1          `before` rid2        = rid1 <= rid2
```

Der genaue Mechanismus zum Durchführen der Heuristik wird hier nicht wiedergegeben, da sein Code vor allem aus Verwaltung des Tableaus besteht und diese praktisch identisch in [Man99] zu finden ist.

Lösung

Die Darstellung der Lösungsfunktion nimmt mehrere Seiten in Anspruch; sie wird deswegen gesondert in Abschnitt 7.7 beschrieben.

Aufruf der einzelnen Phasen

Die einzelnen Phasen der Tableauverarbeitung werden fast genau wie in [Sch99a] analog zu [Man99] aufgerufen, weswegen dies hier nicht geschildert wird. Zwei Unterschiede bestehen: Der eine ist, daß die Aufrufe hier großteils monadisch sind, da manche Schleifen- und Expansionsregeln die `DDAction`-Monade benutzen.

Der andere Unterschied betrifft die Behandlung der Expansionsregeln. Wie in Abschnitt 7.4.2 ausgeführt, werden alle Regeln in der Expansionsphase als Hyperregeln vom Typ `HyperExpansionRule DemandNode` verwendet. Definiert sind die einzelnen Regeln jedoch entweder als `ExpansionRule SimpleDemandNode` oder als `ExpansionRule DemandNode`, je nachdem, ob sie nur einzelne Constraints bearbeiten oder den ganzen Knoten. Wir benötigen also zwei Konvertierungsfunktionen.

- `convertSimpleRule` übernimmt die Arbeit für die Regeln des Typs `ExpansionRule SimpleDemandNode`. Mit `VAnd` verknüpfte Ergebnisse der Ursprungsregel werden zusammen in eine Liste geschrieben; `VOr` bedeutet wie bisher eine Tableauverzweigung.

```
> convertSimpleRule :: ExpansionRule SimpleDemandNode
>                   -> HyperExpansionRule DemandNode
> convertSimpleRule rule = \label ->
>   [ (addRest rest . rule . takeConstraint c) label
>    | (c, rest) <- permutList (constraints label) ]
>   where
>   takeConstraint c label = label {constraints = Just c}
>   addRest rest (VAnd labels@(l1:ls))
>     = VJust $ l1 { constraints =
>                    concat (map getConstraints labels) ++ rest }
>            -- ein VAnd wird zu einem Knoten zusammengefasst
>   addRest rest vL = fmap (addRest' rest) vL
```

```
>   addRest' rest label
>     = label {constraints = getConstraints label ++ rest}
>   getConstraints = maybeToList . constraints
```

Das auch in der folgenden Funktion benutzte `permutList` erzeugt dabei für eine Liste `xs` eine Liste von Paaren `(x,rs)`, so daß `x` jeweils einmal für jedes Element von `xs` steht und `rs` die dazugehörige Liste der restlichen Elemente ist.

- Die anderen Regeln sind ebenfalls noch nicht als Hyperregeln formuliert, behandeln dafür aber immer nur das erste Constraint aus der Liste. `generalizeRule` verallgemeinert sie, so daß jedes Constraint behandelt werden kann.

```
> generalizeRule :: ExpansionRule DemandNode
>                -> HyperExpansionRule DemandNode
> generalizeRule rule = \label ->
>   [ rule (label {constraints = c:rest})
>    | (c, rest) <- permutList (constraints label) ]
```

7.6 Analysen höherer Ordnung

Die Funktion

```
> analyseFunc :: [Name] -> AddedVarDem -> DemandTerm -> DDAction DemandTerm
```

führt eine Analyse höherer Ordnung durch, wie sie in Kapitel 6 beschrieben sind. Im Aufruf `analyseFunc oV v d` steht `d` für die Lösung von $f\ y \in C_2$, wenn es im Kalkül um die Anfrage $f \in C_1 \stackrel{*}{\to} C_2$, $* \in \{\forall, \exists\}$ ging. `v` enthält Informationen über die Einführung der Variable y, und `oV` ist die Liste der Namen der freien Lambda-Variablen von der zugehörigen Anfrage inklusive y.

Zuallererst wird zwischen $\stackrel{\forall}{\to}$ und $\stackrel{\exists}{\to}$ unterschieden.

```
> analyseFunc oV v@(_,ad,FORALL) d = analyseForall oV ad d
> analyseFunc oV v@(_,ad,EXISTS) d = analyseExists oV ad d
```

7.6.1 Die $\stackrel{\exists}{\to}$-Analyse

Die Analyse im $\stackrel{\exists}{\to}$-Fall beansprucht nur wenige Zeilen.

```
> analyseExists oV c1 d = do c1consi <- isConsistentF c1
>                            case c1consi of
>                              Just False -> return emptyDemand
>                              _          -> result
>   where
>   n = length oV - 1
>   result = jecProj Sound [1..n] $
>            mkDIntersection [ d, topWithC1 ]
>   topWithC1 = mkDJec (replicate n (DName "Top") ++ [c1])
```

In der Theorie wurden keine Projektionen auf Schnitten angegeben. Die Implementierung gibt deshalb einen leeren Kontext zurück. Hätten wir eine Projektion, die auch auf Schnitten definiert ist, müßte sie jedenfalls so aufgerufen werden.

7.6.2 Die $\overset{\forall}{\to}$-Analyse

Bei Analysen mit $\overset{\forall}{\to}$-Kontexten legen wir zuerst eine maximale Rekursionstiefe fest:

```
> analyseForallRecursionDepth = 3
```

Schritt 1

Die Analyse selbst behandelt erst den Fall, daß C_1 inkonsistent ist.

```
> analyseForall oV c1 d
>   = do c1consi <- isConsistentF c1
>        case c1consi of
>          Just False -> return $ mkDJec (replicate n (DName "Top"))
>          _  -> anaForallStep2 analyseForallRecursionDepth (n+1) c1 d (n+1)
>   where
>   n = length oV - 1
```

Schritt 2

Der zweite Schritt des Verfahrens 6.2 wird `anaForallStep2 recLevel m c1 g p` aufgerufen. `c1`, `g` und `p` entsprechen den Parametern C, G, p des Theorieschrittes. `m` ist die Stelligkeit von `g`, und mit `recLevel` ist eine Rekursionsebene angegeben, da es dieser Schritt ist, der rekursiv aufgerufen werden kann.

```
> anaForallStep2 :: Int -> Arity -> DemandTerm -> DemandTerm -> Int
>                -> DDAction DemandTerm

> anaForallStep2 recLevel m c1 g p
>   = if botcons c1 then botconsresult
>                   else normalresult
>   where
>   botcons d = recGen f d `startingWith` undefined
>     where f d | (isBot .||. isConstr .||. isJec) d = and
>           f _                             = const False
>   botconsresult = return $ DWhere whereS (whereT, g)
>   whereS = (mkDJec . tack_y_s) (delete p [1..m])
>   whereT = mkDJec (tack_y_s [1..p-1]
>                    ++ [c1]
>                    ++ tack_y_s [p+1..m])
>   tack_y_s = map DVar . zipWith (++) (repeat "y_") . map show
>   normalresult = anaForallMain recLevel m c1 g p
```

Nachdem der Fall abgehandelt wurde, daß `c1` nur aus `Bot` und Konstruktoren besteht, wird also mit den gleichen Parametern wie von `anaForallStep2` die Hauptanalyse `anaForallMain` durchgeführt.

Diese wird weiter unten aufgeführt. Vorher besprechen wir verschiedene Teile, die von ihr benutzt werden: Die Bildung von Information über `c1`, die Bildung der Schnitte und Vereinigungen sowie die der theoretischen Abbildung F entsprechende Funktion.

`c1`-Information

Für Einträge t aus der Vereinigung G muß berechnet werden, ob $C \leq t_p$ ist, wobei t_p die p-te Komponente von t (inklusive einem eventuellem `where`) ist. Da diese Information für alle Einträge aus G gebraucht wird und auch an mehreren Stellen verwendet wird, wird deren Berechnung in einer eigenen Funktion ausgeführt. Die gewonnene Information zu einem dieser Einträge t nennen wir kurz `c1`-*Information* und stellen dafür den Datentyp

```
> type DemWithC1Info = (DemandTerm, Maybe Bool)
```

bereit. Ein Paar dieses Typs enthält einen Kontext t sowie dessen `c1`-Information. Die Erzeugung der Paare erfolgt mit

```
> add_c1Info :: DemandTerm -> Int -> DemandTerm -> DDAction DemWithC1Info
> add_c1Info c1 p jec
>   = fmap ((,) jec) (c1 `isIn` pEntry)
>  where
>   pEntry = p_entry p jec
```

Die oben benutzte Funktion `p_entry` erzeugt die p-te Komponente des Kontextes `jec`. Dieser kann ein echter Jec sein oder aber ein in ein `where` verpackter Jec. `p_entry` muß beides berücksichtigen und im `where`-Fall auch ein `DWhere` zurückgeben. Die folgende Funktion stellt diese Funktionalität bereit:

```
> demOnMaybeWhere :: (DemandTerm -> DemandTerm)
>                  -> DemandTerm -> DemandTerm
> demOnMaybeWhere op (DWhere s (t,n)) = (DWhere (op s) (t,n))
> demOnMaybeWhere op d                = op d
```

Eine ähnliche, weiter unten gebrauchte Funktion ist `opOnMaybeWhere`, das ebenfalls in ein `DWhere` hineingreift, aber ein Ergebnis beliebigen Typs liefert:

```
> opOnMaybeWhere :: (DemandTerm -> a) -> DemandTerm -> a
> opOnMaybeWhere op (DWhere s (t,n)) = op s
> opOnMaybeWhere op d                = op d
```

Mit `demOnMaybeWhere` können nun `p_entry` und eine dazu sozusagen komplementäre Funktion `other_entries` definiert werden.

Die dort benutzte Funktion `jecParts` ist an anderer Stelle im Programm definiert und wird nicht zitiert, denn sie arbeitet mit einer Fallunterscheidung ähnlich wie `demOnMaybeWhere`; sie erzeugt für einen Jec eine Liste seiner Teilkontexte und für jeden anderen Kontext `c` die Liste `[c]`.

```
> p_entry p = demOnMaybeWhere p_entry'
>   where p_entry' d = (jecParts d)!!(p-1)
```

```
> other_entries p = demOnMaybeWhere other_entries'
>   where other_entries' = mkDJec . leaveOut p . jecParts
>         leaveOut 1 (x:xs) = xs
>         leaveOut n (x:xs) = x : (leaveOut (n-1) xs)
>         leaveOut n xs = error ("AnalyseForall.other_entries.leaveOut "
>                                ++show n++" "++show xs)
```

Schnitte / Vereinigungen

Die Ergänzung von G durch zusätzliche Kontexte, die durch Schneiden und Vereinigen von Teilkontexten entstehen, wird von `addIntersToJecs` vorgenommen. Diese Funktion arbeitet auf einer Liste von Kontexten, die zusammen die Einträge von G darstellen.

```
> addIntersToJecs :: DemandTerm -> [DemWithC1Info] -> Int
>                       -> DDAction [DemWithC1Info]
> addIntersToJecs c1 ds p
>   = fmap (ds ++) (additionalEntries c1 p dmdsToUse)
>   where dmdsToUse
>           = (map fst . filter jecToBeConsidered) ds
>         jecToBeConsidered = (/= (Just True)) . snd
```

`dmdsToUse` steht dort für die Liste der Kontexte, aus der die Schnitte und Vereinigungen (im folgenden „Kombinationen“ genannt) gebildet werden müssen (entsprechend der Indexmenge I aus der Theorie). Für die Bildung dieser Kombinationen wird `additionalEntries` aufgerufen:

```
> additionalEntries :: DemandTerm -> Int -> [DemandTerm]
>                   -> DDAction [DemWithC1Info]
> additionalEntries c1 p
>    = allAddedInters c1 p
>    . renameWhereVars
```

Wie der Typ zeigt, erzeugt die Funktion auch gleich die `c1`-Informationen der Kombinationen.

Der Aufruf von `renameWhereVars` sorgt zunächst dafür, daß alle in G vorhandenen `where`-Bindungen gegebenenfalls so umbenannt werden, daß keine zwei gleichen Variablennamen in verschiedenen G-Einträgen vorkommen. So muß sich `allAddedInters` nicht um Namenskonflikte kümmern. Die Funktion benutzt eine optisch etwas unübersichtliche Akkumulierfunktion, so daß wir ihre Darstellung auslassen. `allAddedInters` erzeugt die Kombinationen auf den so modifizierten Einträgen von G. Dabei geht die Funktion in Phasen vor: Es werden Kombinationen von jeweils $2, 3, \ldots, n$ Einträgen erzeugt, wobei n die Anzahl der Einträge in G ist. In den einzelnen Phasen werden auch `c1`-Informationen der Kombinationen erzeugt; diese werden sogleich benutzt, um unnötige Kombinationen und Berechnungen von `c1`-Informationen zu sparen:

Denn wenn eine Kombination von Einträgen $a_1, \ldots, a_i$ erzeugt wurde und die zugehörige `c1`-Information `Just True` ist, dann muß für keine Obermenge der Einträge $\{a_1, \ldots, a_i\}$ ihre Kombination erzeugt werden. Dies wäre unnötig, da bei allen solchen Kombinationen ebenfalls die `c1`-Information `Just True` wäre. Auch in der Theorie würde die Analyse keine solchen Kontexte erzeugen, wenn die $(a_1, \ldots, a_i)$-Kombination schon selbst ein Eintrag von G wäre, was ja keinen Unterschied in der Konkretisierung macht (Punkt 1 im Beweis von Lemma 6.3).

Außerdem wird innerhalb der Phasen auf Bildung inkonsistenter Schnitte überprüft: Häufig entstehen Schnitte wie $[] \cap (C{:}D)$; Kontexte, die diese enthalten, können ganz ignoriert werden und müssen auch nicht weiter kombiniert werden.

```
> allAddedInters :: DemandTerm -> Int -> [DemandTerm]
>                 -> DDAction [DemWithC1Info]
> allAddedInters c1 p ds
>   = fmap fst $
>   foldM f ([],[]) [nAddedInters c1 p n ds | n <- [2..length ds]]
>   where
>   f (alreadyAddeds, notToUses) nAddFunc
>     = do (newToAdds, newNotToUses) <- nAddFunc notToUses
>          return (newToAdds ++ alreadyAddeds, newNotToUses)
```

`nAddedInters` ist für die einzelnen Phasen zuständig. Neben den Kombinationen und `c1`-Informationen werden dort auch die Indizes für nicht mehr benötigte Kombinationen berechnet und auf inkonsistente Schnitte überprüft. Dies hier darzustellen, würde zu weit ins Detail führen.

Zu erwähnen ist aber noch das eigentliche Kombinieren einer Liste von Einträgen; das erledigt eine Hilfsfunktion:

```
>   intersAtPosAux p ls
>     = let (firsts, pos:lasts) = splitAt (p-1) ls in
>     (map mkDIntersection firsts) ++ [mkDUnion pos] ++
>     (map mkDIntersection lasts)
```

Die Funktion F

`anaForallMainMap` entspricht der Funktion F des Verfahrens 6.2. Der Aufruf erfolgt auf den gleichen Parametern wie `anaForallStep2` — `recLevel`, `m`, `c1`, `p`, `g` — sowie einem Argument vom Typ `DemWithC1Info`, das einem der Einträge von G entspricht. Die Implementierung ist relativ direkt der Theorie nachempfunden, wobei die Fallunterscheidung hier zuerst die `c1`-Information abfragt, damit man sie mit möglichst wenig Abfragen formulieren kann. `n'`, `t'` und `n''` stehen für N^*, T^* und N^{**}. Der letzte Fall des `case incl of ...` kommt dann zur Geltung, wenn keine der Bedingungen für die Behandlung von F gegeben ist. Er sichert die Korrektheit unter diesen Umständen.

Man muß hierbei den Fall beachten, wenn die `c1`-Information unklar ist (`Nothing`): Dann kann der entsprechende Eintrag zur Erhaltung der Korrektheit normalerweise nicht in das Ergebnis übernommen werden. Sind dort jedoch Variablen vorhanden, darf mit ihm die allgemeine `where`-Rekursion gemacht werden.

Vom Paar im Ergebnistyp entspricht die linke Komponente dem eigentlichen Ergebnis. Im rechten Teil ist eine eventuell auftretende rekursive Analyse gespeichert, die hier zunächst noch nicht ausgeführt wird, sondern an die Hauptfunktion der Analyse weitergegeben wird, die weiter unten erklärt wird. Dort wird auch näher auf die Verwaltung der Rekursionen eingegangen.

```
> anaForallMainMap :: Int -> Arity -> DemandTerm
>                 -> Int -> DemandTerm -> DemWithC1Info
>       -> DDAction (DemandTerm, Maybe (Name, DDAction DemandTerm))
> anaForallMainMap recLevel m c1 p g (a,incl)
>   = do sC      <- if not noVarsAtP then specialNConds else (return undefined)
>        c1_conv <- isConventional c1
>        g_conv  <- isConventional g
>        case incl of
>          Just False                  -> return0 emptyDemand
>          Nothing   | noVarsAtP       -> return0 emptyDemand
```

```
>          Just True | noVarsAtP       -> return0 rest
>                    | noVarsInRest    -> return0 restWithoutWhere
>                    | baseConds && sC -> do the_n'' <- n''
>                                           return0 (DWhere restWithoutWhere
>                                                           (t', the_n''))
>          _         | baseConds && c1_conv && g_conv && recLevel > 0
>                         -> do ~(n', nextMonad) <- recursion
>                               return ( DWhere restWithoutWhere
>                                                  (t',n')
>                                      , Just (dDName n',nextMonad))
>          _                        -> return0 emptyDemand
>   where
>   return0 x = return (x,Nothing)
>   (noVarsAtP,noVarsInRest) = paired null (varsAtP, varsInRest)
>   (varsAtP, varsInRest)    = paired demandVars
>                          (pEntryWithoutWhere, restWithoutWhere)
>   (pEntryWithoutWhere, restWithoutWhere)
>                            = paired (opOnMaybeWhere id) (pEntry, rest)
>   pEntry = p_entry p a
>   rest = other_entries p a
>   specialNConds = recGen f n
>       where f d | (isForall .||. isExists .||. isWhere .||. isUnion) d
>                 = const False --soll konventionell sein
>                               --ohne Where und Vereinigungen
>             f _ = and
>   baseConds = length varsAtP == 1
>            && x `notElem` varsInRest
>            && (filter (x `elem`) . map demandVars . jecParts) t
>                 == [[x]]
>   [s,t,n] | isWhere a = demParts a
>           | otherwise = error $ "AnalyseForall.anaForallMainMap: etwas \
>                                 \greift auf nicht vorhandenes Where zu \
>                                 \bei "++show a
>   t' = other_entries p_x t
>   t_xs = [ p | p <- jecParts t , x `elem` (demandVars p) ]
>   d  = equivDemand x c1 pEntryWithoutWhere (head t_xs)
>   recursion
>     = do the_d <- d
>          recursiveAnalysis recLevel (jecArity t) the_d n p_x
>   n'' = jecProj Sound (delete p_x [1..(jecArity t)]) n
>   x   = head varsAtP
>   p_x = 1 + (fromJust . findIndex (x `elem`) . map demandVars . jecParts) t
```

Im Rekursionsfall wird `recursiveAnalysis` aufgerufen, sofern `recLevel > 0` zutrifft; andernfalls wird ein leerer Kontext zurückgegeben. Mehr zu `recursiveAnalysis` ist weiter unten zu finden.

Die Rekursion braucht den Kontext D bzw. `d`, der nach den Ausführungen in Abschnitt 6.1.2 berechnet wird. Das wird von `equivDemand` erledigt. Im Aufruf `equivDemand x c1 e p` entsprechen die Parameter `x`, `c1`, `e`, `p` den Vorkommen von a, C_1, x_{n+1}, $P(a)$ in Gleichung (6.1).

```
> equivDemand x@(DVar x') c1 e p
>   = do d' <- find_d' x c1 e
>        return $ applySubst (S [x' :->: d']) p
```

Die Funktion `find_d'` handelt hierbei die Fallunterscheidungen ab, die nötig sind, um D' zu bestimmen. Hier sei der Fall für Vereinigungen angegeben, die anderen sehen ähnlich aus.

```
> find_d' x c (DUnion uset)
>   = do cu <- complement u
>        find_d' x (mkDIntersection [c,cu]) v
>   where us = set2List uset
>         v = head $ filter contains_x us
>         u = mkDUnion $ delete v us
>         contains_x = not . null . demandVars
```

Die Hauptfunktion

Wir kommen nun zur Funktion `anaForallMain`; der Hauptfunktion nach Abhandlung der einfachen Fälle. Sie veranlaßt

1. die Erzeugung der `c1`-Information für jeden Eintrag,
2. die Erweiterung von G durch die Schnitt-/Vereinigungs-Kontexte,
3. den Aufruf von `anaForallMainMap` auf jedem nun vorhandenen Eintrag,
4. die Definition eines Namens für die Analyse
5. und gegebenenfalls den Aufruf von Rekursionsanalysen.

`unionParts` gibt dabei analog zu `jecParts` die Liste der Teilkontexte einer `DUnion` zurück, und eine Liste mit den Gesamtkontext sonst.

```
> anaForallMain :: Int -> Int -> DemandTerm -> DemandTerm -> Int
>               -> DDAction DemandTerm

> anaForallMain recLevel m c1 g p
>   = do gjec <- lookupNameDef g
>        gjecsAug <- mapM (add_c1Info c1 p) (unionParts gjec)
>        analyses <-
>           mapMM (anaForallMainMap recLevel m c1 p g)
>                 (extendedByInters gjecsAug)
>        let (unionlist, recursiveAnas) = unzip analyses
>            analysis = mkDUnion unionlist
>            recursives = [ x | Just x <- recursiveAnas ]
>        pokeDef (anaForallStep2Name recLevel m c1 g p,
>                 analysis)
>        doRecursives recursives
>        return analysis
>   where
>     extendedByInters jecsAug = addIntersToJecs c1 jecsAug p
>     mapMM :: Monad m => (a -> m b) -> m [a] -> m [b]
>     mapMM f = (>>= (mapM f))
>     doRecursives xs = mapM f xs
>       where
>       f (name, monad) = do anaNameDef <- lookupNameDefMaybe (DName name)
>                            case anaNameDef of
>                              Just _  -> return ()
```

```
>                               Nothing -> do result <- monad
>                                            pokeDef (name, result)
```

Die Rekursionen werden nur dann ausgeführt, wenn über ein `lookupNameDefMaybe` festgestellt wird, daß diese noch keinem definierten Namen entsprechen. Dieser Mechanismus verhindert, daß eine Analyse sich selbst im Rekursionsfall erneut ausrechnet. Das wird gleich deutlicher werden mit dem hier nachgelieferten Code des in `anaForallMainMap` erzeugten Rekursionsfalles:

```
> recursiveAnalysis :: Int -> Int -> DemandTerm -> DemandTerm -> Int
>          -> DDAction (DemandTerm, DDAction DemandTerm)
> recursiveAnalysis recLevel new_m d n p_x
>   = do ndef <- lookupNameDef n
>        let anaName  = anaForallStep2Name (recLevel-1) new_m d n p_x
>            analysis = anaForallStep2     (recLevel-1) new_m d n p_x
>        return ( DName anaName
>               , analysis )
```

Diese Funktion gibt sowohl den Namen der rekursiven Analyse zurück als auch die rekursive Analyse selbst; aber diese ist als Monade verpackt, so daß sie noch nicht ausgeführt wird. Das erfolgt ja erst in `anaForallMain`. Erst dort wird die Ursprungsanalyse als Name im Monadenzustand definiert, so daß eine Rekursion auf (direkt oder indirekt) sich selbst erkannt werden kann und nicht neu berechnet wird.

Es bleibt uns noch, den Namen einer Analyse anzugeben. Sie wird eindeutig bestimmt durch C, G und p. Das Rekursionsniveau dient ohnehin nur der Approximation und wird daher nicht berücksichtigt; dabei ist zu betonen, daß schlechtere Approximationen (also die mit dem kleineren `recLevel`) für die gleiche Analyse wegen des gleichen Namens nie berechnet werden.

Da hier eine eindeutige Angabe jeder Analyse (genauer gesagt: der in `anaForallStep2` stattfindenden Analyse) möglich ist, auch wenn sie sich in einer Rekursion nicht direkt auf Kalkülanfragen beziehen, verzichten wir der Einfachheit halber auf zusätzliche Namen wie f-`C2`. Um diesen Namen zu bilden, bräuchte man eine zusätzliche Information innerhalb von `anaForallMain`, die den ursprünglichen aus der Kalkülanfrage stammenden Kontext bereitstellt.

```
> anaForallStep2Name recLevel m c g_name p
>   = "Analyse("++show c
>          ++","++show g_name
>          ++","++show p++")"
```

Der Name hat also das folgende Aussehen: „Analyse(C, G, p)“.

7.7 Lösungen

Nun behandeln wir die Erzeugung der Kontextinformation eines Tableaus. Bei der Lösung muß beachtet werden, daß sich Schleifen auf innere Knoten des Tableaus beziehen, die einen anderen Kontext darstellen können als der Wurzelknoten. Deshalb werden Kontexte auch für alle Knoten, auf die sich Schleifen beziehen, generiert und in den Kontextdefinitionen abgelegt.

Auch die Liste der freien Variablen in den inneren Knoten kann sich von derjenigen der Wurzel unterscheiden. Dieser Parameter beeinflußt aber die Bildung des Kontextes, da er die Stelligkeit und die innere Reihenfolge des zu erzeugenden Jecs angibt. Der Kontext eines Blattes ist also von diesem Parameter abhängig zu machen.

Außerdem müßen Anwendungen der Regel `addArg` berücksichtigt werden, indem an entsprechender Stelle die Analyse aufgerufen wird.

Die Kontextinformation eines Blattes wird von `demandInLeaf` gebildet. Es unterscheidet zwischen Schleifenknoten und gelösten Blättern. Vorausgesetzt ist, daß der Aufruf tatsächlich auf einem gelösten (bzw. Schleifen-) Blatt erfolgt. Der Parameter `fVars` steht dabei für die oben erwähnte Liste von freien Variablen.

```
> demandInLeaf (TLeaf (DemandNode { addLabel = Just (Loop f mL)
>                                 , ccvs = c }))
>              fVars
>   = freeVars2WhereTop $
>     singleFreeVars2Top $
>     applySubst theCCVs (DWhere (applySubst boundCCVs s) (t,n))
>   where ws = f fVars
>         DWhere s (t,n) = ws
>         boundCCVs  = restrict theCCVs' (boundDemandVarNames ws)
>         theCCVs    = S [ x :->: valCCV x | (LVar x) `In` d <- c ]
>         theCCVs'   = S [ x :->: valCCV' x | (LVar x) `In` d <- c ]
>         valCCV x   = mkDIntersection [ d | (LVar y) `In` d <- c, y==x ]
>         valCCV' x  = mkDIntersection [DVar x, valCCV x]

> demandInLeaf (TLeaf (DemandNode { constraints = [], ccvs = c,
>                                   sigma = sigma }))
>              fVars
>   = freeVars2WhereTop $
>     singleFreeVars2Top $
>     mkDJec [ mkDIntersection (values x) | x <- fVars ]
>   where values x   = [valCCV x, valSubst x]
>         valCCV x   = mkDIntersection [ d | (LVar y) `In` d <- c, y==x ]
>         theCCVs    = S [ x :->: valCCV x | (LVar x) `In` d <- c ]
>         valSubst x = ( applySubst theCCVs
>                      . coerceLambdaDemand
>                      . applySubst sigma)
>                      (LVar x)
```

Die Funktion `singleFreeVars2Top` ersetzt Variablen, die durch keine Substitutionen und Constraints beschränkt sind, durch `Top`. `freeVars2WhereTop` kümmert sich darum, mehrfach vorkommende Kontextvariablen durch ein `where...` $=$ (`Top`, ... , `Top`) zu binden.

```
> singleFreeVars2Top d = let (ns,f) = demandVarNamesAndFunc d
>                        in f $ map maybeTop ns
>   where maybeTop name
>           = if manyElems (filter (==name) (demandVarNamesMulti d))
>             then DVar name
>             else DName "Top"

> freeVars2WhereTop d
>   = let vs = demandVars d
>         tops = map (const (DName "Top")) vs
>     in case vs of [] -> d
>                   _  -> DWhere d (mkDJec vs, mkDJec tops)
```

Die Funktion `collectSolution` organisiert die Kontextlösungen von Tableaus. Sie akzeptiert eine Liste `ls` von Tableauknoten (genauer: Tableaus, also Knoten mit ihren Untertableaus) und gibt ein Paar zurück, das aus der Liste der zugehörigen, von einer Variablenliste abhängigen Kontexte besteht sowie aus einer Liste der Knoten, auf die sich Schleifen aus `ls` beziehen. Für letztere müssen nämlich Kontextdefinitionen erzeugt werden, wenn `collectSolution` an den entsprechenden Knoten vorbeikommt.

```
> collectSolution :: [DemandTableau]
>     -> DDAction ([Name] -> [DemandTerm],
>                  [DemandNode])

> collectSolution [] = return (const [],[])
```

Die Knoten in der Liste des Arguments werden nun nacheinander abgearbeitet.

Von Blättern, die eine Schleife enthalten, werden außer der Kontextinformation auch die Bezugsknoten der Schleife gemerkt.

Die dabei auftretende Funktion `mapPairOnPair` wendet ein Funktionenpaar komponentenweise auf ein Argumentpaar an. `joinSubRes` fügt die Lösung an das rekursiv gefundene Restergebnis an und wird weiter unten erklärt.

```
> collectSolution ( l@(TLeaf (DemandNode
>                              { addLabel = Just (Loop _ loopingNode) }))
>                 : rest)
>   = fmap (mapPairOnPair ( joinSubRes (:) (demandInLeaf l) ,
>                           (loopingNode :) ))
>          (collectSolution rest)
```

Blätter ohne Schleife geben nur ihren Kontext zurück.

Das hier benutzte `isContributingLeaf` ist ein Prädikat ähnlich einer Funktion aus [Sch99a], welches auf den von `demandInLeaf` akzeptierten Blättern `True` ergibt. `mapOnFst` wendet eine Funktion auf die erste Komponente (hier die rekursiv ermittelte Lösung von `rest`) eines Argumentpaares an und gibt das ganze Paar zurück.

```
> collectSolution (l@(TLeaf lb): rest)
>   | isContributingLeaf l
>       = fmap (mapOnFst (joinSubRes (:) (demandInLeaf l)))
>              (collectSolution rest)
>   | otherwise
>       = collectSolution rest
```

Bei Verzweigungen wird zuerst der Fall abgehandelt, daß der Kindknoten durch `addArg` entstanden ist. In diesem Fall wird eine Analyse auf der rekursiv gebildeten Lösung des Untertableaus aufgerufen. Schleifenknoten aus dem Untertableau müssen nicht weiter gemerkt werden, da eine Schleife sich nicht über die `addArg`-Grenze erstrecken kann.

Es wird der Ausdruck `Result subsolName subsol` benutzt; dieser hat den Typ `ResultExpr` aus einem Modul aus [Sch99a]. Ein solcher Ausdruck stellt eine Kontextdefinition dar, deren Name mit dem Konstruktor `DConstraints` gebildet ist, die also ein Ergebnis einer Kalkülanfrage (hier aus dem Untertableau) ist.

```
> collectSolution (b@(TBranch bs lb):rest)
>   | childAddsArg
>     = do subResult <- solveTableau child
>          pokeResult subResult
>          let Result subsolName subsol = subResult
>          sol <- analyseFunc oV (head $ addedVarDems ch_lb)
>                                (DConstraints subsolName)
>          fmap (mapOnFst (joinSubRes (:) (const sol)))
>               (collectSolution rest)
>   where
>   hasSingleChild = length bs' == 1
>   bs'   = variousToList bs
>   child = head bs'
>   ch_lb = labelOfTabNode child
>   childAddsArg = if hasSingleChild then
>                        ruleID ch_lb == AddArg
>                      else False
>   oV = freeVarsOfNode ch_lb
```

In allen anderen Verzweigungen wird zunächst rekursiv die Lösung der Teiltableaus berechnet. Die Vereinigung dieser Lösungen (durch `coll2Demand`) entspricht der Lösung des Knotens. Dann wird anhand der zweiten Ergebniskomponente überprüft, ob man sich an einem Knoten befindet, auf den sich eine Schleife bezieht; in diesem Fall wird dessen Lösung mit `pokeResult` in die Kontextdefinitionen aufgenommen.

```
> collectSolution (b@(TBranch bs lb):rest)
>     = do (f, branchLoops) <- collectSolution the_bs
>          if lb `elem` branchLoops
>            then let branchSol = coll2Demand (f (fVars lb))
>                     branchSolName = constraints lb
>                 in pokeResult (Result branchSolName branchSol)
>            else return ()
>          fmap (mapPairOnPair (joinSubRes (++) f, (branchLoops ++)))
>               (collectSolution rest)
>   where the_bs = variousToList bs
>         fVars = freeVarsOfNode

> coll2Demand :: [DemandTerm] -> DemandTerm
> coll2Demand = mkDUnion
```

`collectSolution` hat zwar eine Liste von Knoten als Parameter, behandelt aber immer nur den ersten. Die in jedem Unterfall erwähnte Funktion `joinSubRes` dient dazu, den Kontext dieses Knotens mit den rekursiv ermittelten Lösungen der anderen Knoten in eine Liste zu bringen. `op` ist dabei immer `(:)` oder `(++)`, je nachdem, ob nur eine Lösung oder eine Liste von Lösungen angefügt werden muß.

```
> joinSubRes op fstRes sndRes oVars
>   = fstRes oVars `op` sndRes oVars
```

Aufgerufen wird `collectSolution` von `solveT`; dort wird auch die Lösung des Gesamttableaus definiert.

```
> solveT :: [Name] -> DemandTableau -> DDAction ResultExpr
> solveT fVars tableau
>   = do ~(f,_) <- collectSolution [tableau]
>        let sol = f fVars
>            result = Result cs (coll2Demand sol)
>        pokeResult result
>        return result
>   where cs = constraints (labelOfTabNode tableau)
```

`solveT` ist immer noch von einem Parameter `fVars` abhängig. Dieser ist für ein Gesamttableau immer die Liste der freien Variablen in der Wurzel.

Die Hauptfunktion ist damit

```
> solveTableau :: DemandTableau -> DDAction ResultExpr
> solveTableau tableau = solveT oV tableau
>   where oV = freeVarsOfNode (labelOfTabNode tableau)
```

7.8 Andere Module

Dieser Abschnitt behandelt noch nicht besprochene Teile der Implementierung, die nötig sind, um das Bild des Programms abzurunden, oder weil sie Funktionen definieren, die in früheren Abschnitten benutzt worden sind.

7.8.1 Das Hauptprogramm

Die ganze Analyse läßt sich aus einem Hauptmodul heraus aufrufen, das sich an das entsprechende Modul aus [Sch99a] anlehnt. Es stellt eine interaktive Schnittstelle zum Benutzer dar. So ist es möglich, Dateien zu laden, die ein Λ-Programm bzw. vorgegebene Kontextdefinitionen enthalten. Man kann nun Anfragen eingeben und sich die berechnete Lösung sowie eventuell darin benutzte, neu definierte Namen oder Teilresultate anzeigen lassen.

Zur Ein- und Ausgabe von Anfragen und Lösungen muß eine textuelle Darstellung der zugehörigen Datentypen festgelegt werden. Dies geschieht bei der Eingabe über einen Parser, bei der Ausgabe über die `Show`-Instanz der jeweiligen Datentypen. Beides wurde im wesentlichen aus [Sch99a] übernommen. Die entsprechenden Funktionen befinden sich zwar entgegen der Abschnittsüberschrift nicht im Modul des Hauptprogramms, sondern in eigenen Parsermodulen bzw. in den Modulen der zugehörigen Typdefinitionen. Dennoch gehen wir hier kurz auf die Darstellung ein, denn sie wird auch in Kapitel 8 benutzt.

Die Darstellung von Kernsprachenausdrücken entspricht der Grammatik in Abbildung 3.1. Konstruktoren werden aber auch ohne das redundante `Cons` erkannt; so kann man `{2,2}` statt `Cons{2,2}` schreiben. In ersterer Form werden sie auch ausgegeben.

Ausdrücke vom Typ `DemandTerm` werden im großen und ganzen entsprechend der Symbole geschrieben, die in der theoretischen Definition von Kontexten benutzt werden. Vereinigungen schreibt man `<,...,>`, Schnitte `| /\ ,..., /\ |`, Jecs `[,...,]`. Konstruktoren werden wie bei `LambdaCExpr` geschrieben. Ein Ausdruck S `where` $T = N$ wird genau in dieser Form geschrieben, wobei die Repräsentationen von S und T auch dann in eckigen Klammern stehen, wenn sie keine Jecs sind. `Bot`, `Fun`, Namen und Variablen werden einfach ausgeschrieben. Zu Constraints kommen wir im nächsten Absatz, und schließlich werden die Symbole $\overset{\forall}{\to}$ und $\overset{\exists}{\to}$ mit `-A>` bzw. `-E>` übersetzt.

Das von einem Tableau dargestellte Resultat besteht aus der Liste der Constraints seiner Wurzel zusammen mit der Lösung des Tableaus. Die Schreibweise dafür wird mit einem Beispiel klar. Der Anfrage `const` $x \in$ `Fin`$\overset{\forall}{\to}$`Inf` mit dem Ergebnis `Inf` entspricht die Darstellung

```
[const x <- Fin -A> Inf] -> Inf
```

Diese Form betrifft Ausgaben des Hauptprogramms. Als Eingabe genügt das einzelne Constraint, das die Anfrage darstellt, also in diesem Fall `const x <- Fin -A> Inf`.

7.8.2 Mengen

Die Datenstruktur für Kontexte benutzt für Vereinigungen und Schnitte einen Mengendatentyp. Dessen interne Darstellung von Mengen sind geordnete Listen ohne Duplikate.

```
> data (Eq a, Ord a) => Set a = Set [a]
```

Die darauf definierten Operationen basieren auf effizienten Funktionen aus [PJL91]. Weiterhin ist u.a. eine Gleichheit auf Ausdrücken dieses Typs definiert:

```
> instance  (Eq a, Ord a) => Eq (Set a) where
>   a == b = set2List a == set2List b
```

mit

```
> set2List (Set s) = s
```

Dank der Ordnung in den internen Listen ist der Gleichheitstest also sehr effizient als Listengleichheit implementierbar. Davon profitiert z.B. die Bildung eines Ausdrucks vom Typ `Set DemandTerm` selbst, da dafür ja auf Gleichheit der Argumente getestet wird, um Duplikate auszuschließen.

Kapitel 8

Ergebnisse

Es werden nun beispielhaft einige von der Implementierung durchgeführten Analysen präsentiert. Die Ausgaben des Programms wurde dazu etwas übersichtlicher formatiert. Wir diskutieren die Ergebnisse nach praktischen und theoretischen Gesichtspunkten.

Die Reihenfolge der Komponenten eines Jecs entspricht immer der Reihenfolge der in der Anfrage auftretenden Variablenvorkommen.

8.1 Reine Tableauanalysen

Die Anfragen in diesem Abschnitt kommen ohne Verwendung der Analysen aus Kapitel 6 aus. Es wird aber von den Regeln in Abschnitt 5.2 Gebrauch gemacht.

Die ersten Anfragen sind sehr einfach, und die Ergebnisse zeigen vor allem, daß die Regeln korrekt implementiert sind. Das vierte Beispiel beinhaltet einen Funktionskontext bezüglich einer Funktion mit zwei Argumenten, der implizit auf der rechten Seite geklammert wird: `x -A> (y -A> Bot)`.

```
[apply x y <- Bot] ->
  [y -A> Bot, y] where [y] = Top

[apply x ({2,2} ({1,0} ) bot) <- Bot] ->
  ({2,2} ({1,0} ) Bot) -A> Bot

[const x y z <- Bot] ->
  [z -A> Bot, Top, z] where [z] = Top

[f x y <- Bot] ->
  [x -A> y -A> Bot, x, y] where [x, y] = [Top, Top]
```

Die Anfrage `append` $(p\ x)\ y \in$ `Bot` ist ein Beispiel dafür, daß der Kalkül Informationen liefern kann, die auf den ersten Blick eventuell nicht vermutet werden. So ist p nicht nur dann kritisch, wenn es x auf einen Term in `Bot` abbildet, sondern auch, wenn $p\ x \equiv_c$ `[]`.

```
[append (p x) y <- Bot] ->
  <[x -A> Bot, x, Top] where [x] = Top,
   [x -A> ({1,0} ), x, Bot] where [x] = Top>
```

Wir die Anfrage `filter` $p\ xs \in$ `Bot` gestellt, so taucht in dem Teil des Tableaus, bei dem xs mit $y{:}ys$ substituiert wird, der folgende Knoten auf (`f-bot` sei der Name des Resultates der Anfrage):

$$\textit{loop!}\ (p, y{:}ys)\ \texttt{where}\ (p, ys) = \texttt{f-bot}, p \in y \overset{\forall}{\to} \texttt{Cons\{1,0\}}, xs \mapsto y{:}ys$$

Hier ist p also sowohl durch ein `where` gebunden als auch durch ein eigenes CCV beschränkt.

Für dieses Blatt kann der Kalkül noch keine Lösung aufbieten. Eine intuitive Möglichkeit, die Lücke zu schließen, besteht darin, beide Beschränkungen für p in einem Schnitt innerhalb des `where`-Ausdrucks zusammenzufassen zu einer Lösung

$$(p \cap y \overset{\forall}{\to} \texttt{Cons\{1,0\}}, y{:}ys)\ \texttt{where}\ (p, ys) = \texttt{f-bot}; y = \texttt{Top}.$$

Ob das korrekt ist, ist an anderer Stelle zu klären. In den Programmcode dieser Arbeit wurde dieser Ansatz experimentell eingebaut, so daß folgendes Resultat berechnet wird:

```
[filter p x <- Bot] ->
  <[Top, Bot],
   [a_0 -A> Bot, ({2,2} a_0 Top)] where [a_0] = Top,
   [[|(a_0 -A> ({1,0} )) /\ p|, ({2,2} a_0 b_0)]
     where [p, b_0] = filter p x <- Bot]
    where [a_0] = Top>
```

Die nächsten Ergebnisse lassen subtile Unterschiede erkennen, die durch Striktheit verursacht werden. `uncurry_strict` ist „strikt im zweiten Argument“, d.h. es wertet sein zweites Argument immer zu WHNF aus (durch eine `case`-Anwendung). `uncurry` tut das nicht in jedem Fall. Man sieht am Unterschied in der jeweils ersten Zeile der Ergebnisse, daß `uncurry_strict` in mehr Fällen Funktionswerte in γ(`Inf`) hat als `uncurry`. Das liegt daran, daß für $x \equiv_c$ `bot` auch `uncurry_strict` $f\ x \equiv_c$ `bot` ist.

```
[uncurry f x <- Inf] ->
  <[Bot -A> Bot -A> Inf, Bot],
   [_x_1 -A> _x_0 -A> Inf, ({1,2} _x_1 _x_0)]
     where [_x_1, _x_0] = [Top, Top]>

[uncurry_strict f x <- Inf] ->
  <[Top, Bot],
   [a_0 -A> b_0 -A> Inf, ({1,2} a_0 b_0)]
      where [a_0, b_0] = [Top, Top]>
```

Umgekehrt sieht es bei den gleichen Anfragen bezüglich `Fin` aus: `uncurry_strict` liefert weniger Lösungen, da bei x ohne WHNF der Funktionswert, wie oben erwähnt, auch keine WHNF hat und nicht in γ(`Fin`) liegen kann.

```
[uncurry f x <- Fin] ->
  <[Bot -A> Bot -A> Fin, Bot],
   [_x_1 -A> _x_0 -A> Fin, ({1,2} _x_1 _x_0)]
     where [_x_1, _x_0] = [Top, Top]>

[uncurry_strict f x <- Fin] ->
  [a_0 -A> b_0 -A> Fin, ({1,2} a_0 b_0)]
    where [a_0, b_0] = [Top, Top]
```

8.2 Ergebnisse von Analysen höherer Ordnung

Die folgenden Resultate sind mit Hilfe der Analysen aus Kapitel 6 entstanden.

Manche Anfragen haben ein sehr einfaches Ergebnis. Z.B. müssen hier keine `where`-Ausdrücke berücksichtigt werden:

```
[const x <- Fin -A> Inf] ->
  Inf

[flip const x <- Inf -A> Botelem] ->
  Top
```

Von den restlichen Analyseergebnissen in diesem Kapitel beziehen sich die meisten auf Kontextnamen (die Namen der Analysen), die neu definiert werden. Die Definitionen dieser Namen sind immer in den Zeilen direkt unter dem eigentlichen Resultat aufgeführt und sind als Paar vom Typ `DNameDef` notiert.

Wir konzentrieren uns auf $\overset{\forall}{\to}$-Analysen, da die Konkretisierung von $\overset{\exists}{\to}$-Kontexten so kontraintuitiv ist und jene daher von recht zweifelhaftem Wert sind. Da unsere Projektionen nicht auf Schnitten definiert sind, würde außerdem die Implementierung für jede Analyse bezüglich $\overset{\exists}{\to}$-Kontexten nur den leeren Kontext ausgeben.

Die erste Analyse wurde schon in einem früheren Kapitel als Beispiel verwendet:

```
[append xs <- Fin -A> Inf] ->
  <Bot,
   [({2,2} Top zs_1)] where [zs_1] = Analyse(Fin,[append xs x_0 <- Inf],2)>
("Analyse(Fin,[append xs x_0 <- Inf],2)",
  <Bot,
   [({2,2} Top zs_1)] where [zs_1] = Analyse(Fin,[append xs x_0 <- Inf],2)>)
```

Die Methodik der Namensvergabe in der Implementierung führt hier zu einem Unterschied zur Theorie, die nur einen statt zwei Kontextnamen benötigt. Das praktisch errechnete Resultat ist aber dennoch semantisch gleichwertig. Auch wenn das Resultat effektiv doppelt da steht, werden die Berechnungen nicht doppelt ausgeführt; es wird nur ein und dasselbe Resultat unter zwei Namen gespeichert.

Bei der folgenden Analyse sieht man ein redundantes `{2,2} Bot Top`. Dies entspricht der Theorie und läßt sich nicht ohne weiteres vermeiden.

`Botelem` steht dabei für den Kontextnamen mit der Definition $\langle$`Bot,Bot:Top,Top:Botelem`$\rangle$.

```
[append xs <- Inf -A> Botelem] ->
  <Bot,
   ({1,0} ),
   ({2,2} Bot Top),
   [({2,2} Top zs_1)] where [zs_1] = Analyse(Inf,[append xs x_0 <- Botelem],2)>
("Analyse(Inf,[append xs x_0 <- Botelem],2)",
  <Bot,
   ({1,0} ),
   ({2,2} Bot Top),
   [({2,2} Top zs_1)] where [zs_1] = Analyse(Inf,[append xs x_0 <- Botelem],2)>)
```

Bis jetzt waren die Analysen dieses Abschnitts alle korrekt und vollständig. Die folgende Analyse aber ist extrem unvollständig. Vollständig *wäre* ein Ergebnis `Inf`. (`flipappend` ist analog zu `append` definiert, nur mit umgedrehter Argumentreihenfolge.)

```
[flipappend x <- Fin -A> Inf] ->
  [|Inf /\ x|] where [x] = Analyse(Fin,[flipappend x x_0 <- Inf],2)
("Analyse(Fin,[flipappend x x_0 <- Inf],2)",
  [|Inf /\ x|] where [x] = Analyse(Fin,[flipappend x x_0 <- Inf],2))
```

Dies entspricht der Definition `flipapp-Inf-Fin` $=$ (`Inf` $\cap\, ys$) `where` $ys =$ `flipapp-Inf-Fin`. Dessen Konkretisierung ist aber leer!

Wie kommt es dazu? Benutzt wird das Ergebnis

$$\texttt{flipapp-Inf} = \left\langle \begin{array}{l} (\texttt{Top},\texttt{Bot}), \\ (\texttt{Inf},\texttt{[]}), \\ (ys,\texttt{Top:}zs) \text{ } \texttt{where} \text{ } (ys,zs) = \texttt{flipapp-Inf} \end{array} \right\rangle,$$

was noch vollständig ist. Nun ist `Fin` ist in keiner der rechten Komponenten enthalten. Also werden Schnitte und Vereinigungen gebildet; der einzige relevante Kontext, mit dem dann eine rekursive Analyse gemacht wird, ist

$$(\texttt{Inf} \cap ys, \langle \texttt{[]}, \texttt{Top:}zs \rangle) \text{ } \texttt{where} \text{ } (ys,zs) = \texttt{flipapp-Inf}.$$

Die Konkretisierung der Analyse wird dann leer, weil keine Einträge zum Verankern der Rekursion vorhanden sind, im Gegensatz zu den obigen Analysen, bei denen immer z.B. ein `Bot` mit im Lösungkontext stand.

Die auf jedem einzelnen Eintrag der Vereinigung laufenden Analysen sind gewissermaßen für sich vollständig; diese Eigenschaft geht erst im Zusammenspiel dieser Einträge verloren. Hier stützt sich die rekursive Analyse auf eine nicht vorhandene, beliebige andere nichtrekursive Analyse.

Man kann sich vorstellen, daß man den Kalkül auch das Ergebnis `flipapp-Inf-Fin` $=$ `Inf` $\cap$ `flipapp-Inf-Fin` erzeugen lassen könnte. Bei dieser Formulierung ist das gewünschte Ergebnis `Inf` immerhin intuitiv ablesbar: ein Kontext, der ausschließlich mit sich selbst (quasi!) geschnitten wird, könnte ruhig als Schnittergebnis so bleiben, wie er ist. Dabei ist zu betonen, daß η(`flipapp-Inf-Fin`) als kleinster Fixpunkt von Δ leer ist, aber auch η(`Inf`) ein Fixpunkt der zugehörigen Gleichungen ist; nur eben nicht der kleinste. (Das gilt genauso für die erste Definition von `flipapp-Inf-Fin` oben!)

Um sich dieses Problems zu entledigen, könnte man versuchen, die Definition von η zu ändern. Es ist denkbar, für $\eta(C)$ nicht immer die kleinstmögliche Menge zu nehmen, sondern einen Fixpunkt, welcher sich unter Zuhilfenahme eines für Vereinigung *und* Schnitt neutralen Elementes ergibt. Für ein $L \in \Lambda^C$ mit $L = L \cap C$ wäre dann $\eta(L)$ der größte Fixpunkt der Gleichung $\eta(L) = \eta(L) \cap \eta(C)$; und für $M = \langle M, C \rangle$ würde sich analog dazu der kleinste Fixpunkt ergeben. Dies führt aber dazu, daß die Distributivität nicht mehr gilt: $L = \langle L \cap A, B \rangle$ wäre bezüglich der Konkretisierung nicht mehr äquivalent zur Definition $L = \langle L, B \rangle \cap \langle A, B \rangle$. Das macht diesen Lösungsversuch sehr zweifelhaft.

Die nächsten Resultate leiden unter dem gleichen Symptom.

```
[flip append x <- Fin -A> Inf] ->
  [|Inf /\ x|] where [x] = Analyse(Fin,[append x_0 x <- Inf],1)
("Analyse(Fin,[append x_0 x <- Inf],1)",
  [|Inf /\ x|] where [x] = Analyse(Fin,[append x_0 x <- Inf],1))
```

Die obige Analyse ist ähnlich wie die vorhergehende. Da `flip append` x y zu `append` y x reduziert, erfolgt hier die Schleifenbildung einmal nicht mit dem Wurzelknoten des Teiltableaus, das nach Anwendung der Regel AddArg entsteht, sondern mit dem darunter liegenden Knoten mit dem Constraint `append` y $x \in$ `Inf`. Die durch das `flip` geänderte Reihenfolge der Variablen im Knoten macht sich dadurch bemerkbar, daß die Analyse bezüglich der ersten Jec-Komponente erfolgt statt der zweiten.

```
[flip append x <- Inf -A> Botelem] ->
  [x] where [x] = Analyse(Inf,[append x_0 x <- Botelem],1)
("Analyse(Inf,[append x_0 x <- Botelem],1)",
  [x] where [x] = Analyse(Inf,[append x_0 x <- Botelem],1))
```

Die hier auftretende Form x `where` $x = \ldots$ ist durch ein von der Implementierung herausgekürztes `Top` entstanden. Nach der Theorie hätte das Ergebnis die Form (`Top`$\cap x$) `where` $x = \ldots$, also analog zum obigen Symptom mit `Inf`. `Top` wäre auch das korrekte, vollständige Ergebnis.

Das Ergebnis der folgenden Analyse hat ebenfalls eine leere Konkretisierung. Diesmal ist es aber auch vollständig, da es kein xs gibt, bei dem für alle $ys \in \gamma($`Inf`$)$ (oder auch nur für eines dieser ys) die Aussage `append` xs $ys \in \gamma($`Fin`$)$ zutreffen würde.

```
[append xs <- Inf -A> Fin] ->
  [({2,2} Top zs_1)] where [zs_1] = Analyse(Inf,[append xs x_0 <- Fin],2)
("Analyse(Inf,[append xs x_0 <- Fin],2)",
  [({2,2} Top zs_1)] where [zs_1] = Analyse(Inf,[append xs x_0 <- Fin],2))
```

Abschließend betrachten wir die Anfrage `zip` $xs \in$ `Inf`$\stackrel{\forall}{\rightarrow}$`Fin`. Man könnte sich denken, daß `Fin` als Ergebnis herauskommen sollte, da `zip` bei einer endlichen Liste im ersten Argument auch eine endliche Liste als Ergebnis haben sollte. Der Kalkül rechnet jedoch

```
[zip xs <- Inf -A> Fin] ->
  ({1,0} )
```

Das Ergebnis ist vollständig, da ja `zip` ein `case` auch auf das zweite Argument anwendet, und wenn dieses eine genügend kurze Liste aus $\gamma($`Inf`$)$ ist, so ist das ganze Ergebnis ohne WHNF.

Kapitel 9

Zusammenfassung und Ausblick

9.1 Zusammenfassung

In dieser Arbeit wurde die Kontextanalyse aus [Sch99b] um Funktionskontexte erweitert. Dies geschah zum einen durch die Einführung dieser neuen Kontexte, zum anderen durch die Angabe von Verfahrensweisen, die den Umgang der Analyse mit diesen Kontexten erklären. Die erweiterte Analyse wurde schließlich praktisch umgesetzt.

Kapitel 2 führte in das Umfeld der Arbeit ein, indem es einen Einblick in die Grundlagen funktionalen Programmierens gab.

Dann folgten in Kapitel 3 als Basis für den Rest der Arbeit eine Behandlung der Kernsprache, also des Analysegegenstandes, sowie in Kapitel 4 die Definition der Kontexte. Dort wurden auch die Funktionskontexte eingeführt und eine Reihe von Sätzen als Vorbereitung auf die Analyse bewiesen, die teilweise auch einen qualitativen Unterschied dieser Kontexte zu den konventionellen deutlich machten.

Eine Darstellung der Kontextanalyse schloß sich mit Kapitel 5 an. In diesem Rahmen war es bereits möglich, Erweiterungen anzugeben, die unter Benutzung der Funktionskontexte für die Behandlung von Argumenten höherer Ordnung geeignet sind. Die Korrektheit und teilweise auch die Vollständigkeit der Erweiterung konnten gezeigt werden.

Kapitel 6 führte Analyseverfahren ein, die die Kontextanalyse um die Behandlung von Anfragen mit Funktionskontexten erweitern. Ihre partielle Korrektheit konnte weitgehend bewiesen werden, die partielle Vollständigkeit unter Ausschluß einiger Fälle.

In Kapitel 7 wurde die Implementierung des soweit theoretisch Aufgebauten vorgestellt. Die damit durchgeführten Beispielanalysen wurden in Kapitel 8 besprochen. Dabei wurde einerseits die Anwendbarkeit der Analyse vorgeführt. Andererseits wurde klar, daß sie an manchen Stellen noch erweitert werden sollte.

9.2 Ausblick

Das führt uns dazu, noch offene Fragen anzusprechen.

Von Bedeutung ist sicher eine nähere Untersuchung der Bedingungen, unter denen die Aussage, die im Zusammenhang mit der Korrektheit der $\overset{\forall}{\rightarrow}$-Analyse bewiesen wird, tatsächlich eine echte Korrektheit darstellt und ob das vielleicht sogar immer der Fall ist. Dafür gibt es zwar Hinweise, es konnte aber bis jetzt nur vermutet werden.

Eine weitere wichtige Stelle, an der zukünftige Arbeit ansetzen könnte, ist die extreme Unvollständigkeit mancher $\overset{\forall}{\rightarrow}$-Analysen, die in Kapitel 8 zutage getreten ist. Hier könnte möglicherweise das Zusammenspiel der Teilanalysen auf den einzelnen Teilkontexten verbessert werden.

Aufgefallen ist auch, daß Schleifen im Tableau auftreten können, die von der Kontextanalyse, wie sie bisher definiert war, noch nicht mit einem Kontext ausgedrückt werden. An dieser Stelle besteht noch Definitionsbedarf.

Eine ganz andere Frage ist grundlegender: Können Kontexte der Form $A \xrightarrow{\forall} B$ so benutzt werden, daß auch für A Funktionskontexte zugelassen sind, aber trotzdem eine wohldefinierte Semantik gewährleistet ist? Zur Lösung dieses Problems müßte man wohl theoretisch weiter ausholen, als es hier möglich gewesen ist. Wenn sich herausstellt, daß es möglich ist, wäre jedoch eine Verstärkung der Ausdruckskraft der Kontexte die Folge.

Anhang A

Projektionen auf `where`-Kontexten: weitere Fälle

Dieser Anhang erweitert die Definition 4.57 auf Seite 46 und den anschließenden Satz. Die hier definierten Fälle sind allgemeiner als diejenigen aus dem Hauptteil der Arbeit in dem Sinne, daß sowohl der zurückgegebene als auch der verworfene Teil des `where`-Argumentes Variablen enthalten.

Definition A.1 (Erweiterung von Definition 4.57). *Sei* $S = \texttt{c}\ S_1 \dots S_n$. *Alle Komponenten* $S_1, \dots, S_n$ *von* S *seien jeweils eine Variable, enthalten also keine weitere, eigene Struktur. Die* S_i, *die eine Variable aus* V^1 *sind, seien* $S_{i_1}, \dots, S_{i_m}$, *und die restlichen* S_k, *die also eine Variable aus* V^* *sind, seien* $S_{k_1}, \dots, S_{k_r}$.

1. *Falls*

   - *in den Komponenten von* R, *an deren entsprechender Stelle in* S *sich keine Variable aus* V^1 *befindet, genau eine Variable* x *vorkommt, und zwar genau einmal;*
   - x *nirgendwo sonst in* R *vorkommt;*
   - T *ein Jec ist;*
   - x *nur einmal in* T *vorkommt und in der Komponente* $\Pi_{p_x}(T)$ *von* T, *in der* x *steht, keine andere Variable aufgeführt ist*
   - *und* N *ein konventioneller Kontext ohne* `where` *und Vereinigungen ist:*

$$P_{(\texttt{c}\ S_1 \dots S_n)}(\texttt{c}\ C_1 \dots C_n\ \texttt{where}\ T = N) := \begin{cases} (C_{i_1}, \dots, C_{i_m})\ \texttt{where}\ \Pi_{\widehat{p_x}}(T) = \pi_{\widehat{p_x}}(N) \\ \quad \textit{wenn}\ \eta((C_{k_1}, \dots, C_{k_r})\ \texttt{where}\ \Pi_{p_x}(T) = \pi_{p_x}(N)) \neq \emptyset \\ \langle\rangle \quad \textit{sonst} \end{cases}$$

2. *Falls*

   - *in den Komponenten von* R, *an deren entsprechender Stelle in* S *sich keine Variable aus* V^1 *befindet, genau eine Variable* x *vorkommt, nämlich in der Komponente* C_{q_x} *und dort genau einmal;*
   - x *nirgendwo sonst in* R *vorkommt;*
   - T *ein Jec ist;*
   - x *nur einmal in* T *vorkommt und in der Komponente* $\Pi_{p_x}(T)$ *von* T, *in der* x *steht, keine andere Variable aufgeführt ist*

- *und C_{q_x} keine Schnitte enthält:*

$$P_{(\mathtt{c}\ S_1 \ldots S_n)}(\mathtt{c}\ C_1 \ldots C_n\ \mathtt{where}\ T = N)$$
$$:= \begin{cases} (C_{i_1}, \ldots, C_{i_m})\ \mathtt{where}\ \Pi_{\widehat{p_x}}(T) = \pi_U(N) \\ \quad \textit{wenn}\ \eta((C_{k_1}, \ldots, C_{k_r})\ \mathtt{where}\ \Pi_{p_x}(T) = \pi_{p_x}(N)) \neq \emptyset \\ \langle\rangle \quad \textit{sonst,} \end{cases}$$

wobei $U = (x_1^1 \ldots, x_t^1)[(\Pi_{p_x}(T)[x^*/x])/x_{p_x}^1]$,[a] d.h. U veranlaßt, daß alle Komponenten von N zurückgegeben werden außer der p_x-ten Komponente, diese aber wird auf die Struktur von $\Pi_{p_x}(T)$ überprüft.

Im zweiten Fall der Definition darf N auch auf `where`-Ausdrücke verweisen. Da auf N wiederum eine Projektion angewendet wird, gilt in diesem Fall das folgende Lemma nur dann, wenn die Definition eine wohldefinierte Projektion ergibt, d.h. wenn nicht unendlich viele Projektionen rekursiv erzeugt werden müßten. Diese Annahme entspricht der Induktionsverankerung für N.

Lemma A.2. *Lemma 4.54 gilt auch für Kontexte C, die Kontexte der oben angegebenen Formen enthalten.*

Beweis. Wir müssen wieder zeigen:

$$\bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) \overset{!}{\subseteq} \eta_{\Delta^j(\mathbf{0})}(P_S(R\ \mathtt{where}\ T = N)) \overset{!}{\subseteq} \bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \eta(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R))$$

für beide Fälle.

Man beachte die folgenden Gleichheiten (unter Ausnutzung der Voraussetzungen über S):

$$\begin{aligned} & \bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) \\ = & \bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N)} \begin{cases} \eta_{\Delta^j(\mathbf{0})}(\sigma(C_{i_1}, \ldots, C_{i_m})) & \text{wenn } \eta(\sigma(C_{k_1}, \ldots, C_{k_r})) \neq \emptyset \\ \emptyset & \text{sonst} \end{cases} \\ = & \bigcup_{\substack{\sigma:\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N) \\ \wedge \eta(\sigma(C_{k_1}, \ldots, C_{k_r})) \neq \emptyset}} \eta_{\Delta^j(\mathbf{0})}(\sigma(C_{i_1}, \ldots, C_{i_m})) \end{aligned} \tag{A.1}$$

und entsprechend

$$\bigcup_{\sigma:\Lambda_p(\sigma T) \subseteq \eta(N)} \eta_{\Delta^j(\mathbf{0})}(P_S(\sigma R)) = \bigcup_{\substack{\sigma:\Lambda_p(\sigma T) \subseteq \eta(N) \\ \wedge \eta(\sigma(C_{k_1}, \ldots, C_{k_r})) \neq \emptyset}} \eta_{\Delta^j(\mathbf{0})}(\sigma(C_{i_1}, \ldots, C_{i_m})). \tag{A.2}$$

1. Da N ein konventioneller Kontext ohne `where` und Vereinigungen ist, können wir hier einerseits annehmen, daß Lemma 4.54 für N bereits gilt, also $\eta(P_Q(N)) = P_Q(\eta(N))$ für jedes Muster Q. Andererseits sind hier alle einzelnen Konstruktorbestandteile von N unabhängig voneinander, d.h. für jeden Konstruktor $\mathtt{c}$ umfaßt $\eta(N)$ die Kombination $\eta(\mathtt{c}\ P_{1,\mathtt{c}} \ldots P_{\alpha(\mathtt{c}),\mathtt{c}})$ der Projektionen, wie leicht induktiv zu erkennen ist.

[a] Dies ist wie folgt zu lesen: in $(x_1^1 \ldots, x_t^1)$ ist die p_x-te Variable $x_{p_x}^1$ durch $\Pi_{p_x}(T)[x^*/x]$ ersetzt, und dieser Term wiederum ist derjenige, der durch Sternen der Variablen x in $\Pi_{p_x}(T)$ entsteht. Die Notation lehnt sich an diejenige für Λ-Ausdrücke aus Definition 3.4 an.

Es ist

$$\eta_{\Delta^j(\mathbf{0})}(P_S(R \texttt{ where } T = N))$$
$$= \begin{cases} \bigcup\limits_{\sigma:\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T))\subseteq\eta_{\Delta^j(\mathbf{0})}(\pi_{\widehat{p_x}}(N))} \eta_{\Delta^j(\mathbf{0})}(\sigma(C_{i_1},\ldots,C_{i_m})) \\ \quad \text{wenn} \bigcup\limits_{\tau:\Lambda_p(\tau\Pi_{p_x}(T))\subseteq\eta(\pi_{p_x}(N))} \eta(\tau(C_{k_1},\ldots,C_{k_r})) \neq \emptyset \\ \emptyset \quad \text{sonst} \end{cases}$$

Ein τ, das die Gültigkeit der obigen „wenn"-Bedingung bezeugt, ist o.B.d.A. immer gleich einem der σ's, da die Forderungen an τ bzw. σ zwei disjunkte Variablenmengen betreffen, die Werte auf den anderen Variablen also beliebig gewählt werden können:

$$= \bigcup_{\substack{\sigma:\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T))\subseteq\eta_{\Delta^j(\mathbf{0})}(\pi_{\widehat{p_x}}(N)) \\ \wedge\Lambda_p(\sigma\Pi_{p_x}(T))\subseteq\eta(\pi_{p_x}(N)) \\ \wedge\eta(\sigma(C_{k_1},\ldots,C_{k_r}))\neq\emptyset}} \eta_{\Delta^j(\mathbf{0})}(\sigma(C_{i_1},\ldots,C_{i_m})) \tag{A.3}$$

Von den Formeln (A.1) und (A.2) unterscheidet sich der letzte Ausdruck nur in Teilen des Subskriptes der Vereinigung. Zu zeigen ist also

(a) $\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N) \implies \Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) \subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_{\widehat{p_x}}(N)) \wedge \Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N))$ und

(b) $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) \subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_{\widehat{p_x}}(N)) \wedge \Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N)) \implies \Lambda_p(\sigma T) \subseteq \eta(N)$.

Das geht wie folgt:

(a) Sei $\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N)$. Da σT und die syntaktischen Teile davon Kontextmuster sind, für die Lemma 4.54 sicher gilt, ist $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) = \pi_{\widehat{p_x}}(\Lambda_p(\sigma T))$ und weiter

$$\begin{aligned} \pi_{\widehat{p_x}}(\Lambda_p(\sigma T)) &\subseteq \pi_{\widehat{p_x}}(\eta_{\Delta^j(\mathbf{0})}(N)) \\ &\subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_{\widehat{p_x}}(N)) \quad \text{(Induktionsannahme über } N\text{)}. \end{aligned}$$

Das ist die eine Hälfte der zu zeigenden Implikation. Der zweite Teil $\Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N))$ folgt ganz genauso.

(b) Sei $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) \subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_{\widehat{p_x}}(N))$ und $\Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N))$.
Nun ist jeder Λ-Ausdruck in $\Lambda_p(\sigma T)$ ein Jec-Konstrukt, dessen Teile Terme aus $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T))$ und aus $\Lambda_p(\sigma\Pi_{p_x}(T))$ bilden. Umgekehrt entspricht jeder Kombination eines Terms in $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T))$ mit einem aus $\Lambda_p(\sigma\Pi_{p_x}(T))$ genau ein Term aus $\Lambda_p(\sigma T)$. Da die einzelnen Projektionen $\pi_i(\eta(N))$ von N, wie oben bemerkt, voneinander unabhängig sind, d.h. es gibt auch eine 1:1-Entsprechung zwischen den T-strukturierten Ausdrücken in $\eta(N)$ und den Kombinationen von $\Pi_{\widehat{p_x}}(T)$-strukturierten Termen in $\pi_{\widehat{p_x}}(\eta(N)) = \eta(\pi_{\widehat{p_x}}(N))$ und $\Pi_{p_x}(T)$-strukturierten Termen in $\pi_{p_x}(\eta(N)) = \eta(\pi_{p_x}(N))$. Mit den beiden Voraussetzungen ergibt sich dadurch $\Lambda_p(\sigma T) \subseteq \eta(N)$.

2. Der Beweis verläuft anfangs wie im vorigen Fall. Bei einer der zwei Implikationen, die für σ gezeigt werden, müssen wir jedoch die Bedingung zu Hilfe nehmen, die für jedes der σ erfüllt ist bzw. erfüllt sein soll und die im vorigen Fall nicht berücksichtigt zu werden brauchte: nämlich

$$\eta(\sigma((C_{k_1},\ldots,C_{k_r}))) \neq \emptyset. \tag{A.4}$$

Dabei nutzt es uns, daß C_{q_x} ein Kontext ohne Schnitte ist: Die Konsistenz von σC_{q_x} hängt dann nicht von dem von σ eingesetzten Wert für x ab, wie man leicht sieht. Entweder ist $\eta(\sigma C_{q_x}) = \emptyset$ für jedes $\sigma x \in \Lambda_p^C$ oder für keines. Dies bedeutet, daß die Gültigkeit der Formel (A.4) nicht von der verwendeten Substitution σ abhängt.

Wir beweisen die folgenden beiden Implikationen:

(a) $\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N) \implies \Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) \subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_U(N)) \wedge \Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N))$ und

(b)

$$\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) \subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_U(N)) \wedge \Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N)) \wedge \text{(A.4) gilt für } \sigma$$
$$\implies \exists\tau : \Lambda_p(\tau T) \subseteq \eta(N) \wedge \text{(A.4) gilt für } \tau.$$

Und zwar folgendermaßen:

(a) Das Prädikat $R(t, P)$ bedeute, daß der Λ-Ausdruck t zu einem P-strukturierten Ausdruck reduziert.
Sei $\Lambda_p(\sigma T) \subseteq \eta_{\Delta^j(\mathbf{0})}(N)$ und damit auch $\Lambda_p(\sigma T) \subseteq \{t \in \eta_{\Delta^j(\mathbf{0})}(N) | R(t,T)\}$. Da σT und die syntaktischen Teile davon Kontextmuster sind, für die Lemma 4.54 sicher gilt, ist $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) = \pi_{\widehat{p_x}}(\Lambda_p(\sigma T))$ und weiter

$$\begin{aligned}\pi_{\widehat{p_x}}(\Lambda_p(\sigma T)) &\subseteq \pi_{\widehat{p_x}}(\{t \in \eta_{\Delta^j(\mathbf{0})}(N) | R(t,T)\}) \\ &= \{\Pi_{\widehat{p_x}}(t) | t \in \eta_{\Delta^j(\mathbf{0})}(N) \wedge R(t,T) \wedge t \text{ ist } (x_1, \ldots, x_t)\text{-strukturiert}\} \\ &\subseteq \{\Pi_{\widehat{p_x}}(t) | t \in \eta_{\Delta^j(\mathbf{0})}(N) \wedge R(t,U) \wedge t \text{ ist } (x_1, \ldots, x_t)\text{-strukturiert}\}\end{aligned}$$

Ein solches t ist o.B.d.A. selbst U-strukturiert — die Reduktion, die durch $R(t,U)$ gegeben ist, kann nur in der Jec-Komponente von t verlaufen, die ohnehin von $\Pi_{\widehat{p_x}}$ „weggeworfen“ wird.

$$\begin{aligned}&= \{\Pi_{\widehat{p_x}}(t) | t \in \eta_{\Delta^j(\mathbf{0})}(N) \wedge t \text{ ist } U\text{-strukturiert}\} \\ &= \{\Pi_U(t) | t \in \eta_{\Delta^j(\mathbf{0})}(N) \wedge t \text{ ist } U\text{-strukturiert}\} \\ &= \pi_U(\eta_{\Delta^j(\mathbf{0})}(N)) \\ &\subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_U(N)) \quad \text{(Induktionsannahme über } N).\end{aligned}$$

Das ist die eine Hälfte der zu zeigenden Implikation. Der zweite Teil $\Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N))$ folgt wie unter Punkt 1.

(b) Sei $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) \subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_U(N))$. (Die andere Voraussetzung $\Lambda_p(\sigma\Pi_{p_x}(T)) \subseteq \eta(\pi_{p_x}(N))$ wird nicht benötigt).
Es gilt $\Lambda_p(\sigma\Pi_{\widehat{p_x}}(T)) \subseteq \eta_{\Delta^j(\mathbf{0})}(\pi_U(N)) \subseteq \pi_U(\eta(N))$ (die zweite Inklusion ist die Induktionsannahme über N). $\eta(N)$ enthält also U-strukturierte Terme u, deren syntaktische Projektion $\Pi_{\widehat{p_x}}(u)$ jeweils sogar $\Pi_{\widehat{p_x}}(T)$-strukturiert ist. Insgesamt weist jedes u dadurch eine T-Struktur auf. Nach Lemma 4.41 ist $\Lambda(\tau T) \subseteq \eta(N)$ für ein τ.
Außerdem wird dadurch, daß die Gültigkeit der Formel (A.4) von der verwendeten Substitution unabhängig ist, wie wir oben bemerkt haben, diese Ungleichung auch für τ wahr.

$\diamond$

Anhang B

Einige Definitionen von Kernsprachenausdrücken

Es folgt ein Λ-Programm, in dem einige in dieser Arbeit benutzte Superkombinatoren definiert werden.

```
const k x   = k;
id x        = x;
Y f         = f (Y f);
apply f x   = f x;
if p x y    = case p of
                <1> -> y;
                <2> -> x;
uncurry f c = f (fst c) (snd c);
uncurry_strict f c = case c of
                       <1> a b -> f a b;
fst y       = case y of
                <1> x1 x2 -> x1;
snd y       = case y of
                <1> x1 x2 -> x2;
flip f x y  = f y x;
append xs ys = case xs of
                 <1> -> ys;
                 <2> z zs -> Cons{2,2} z (append zs ys);
flipappend xs ys = case ys of
                     <1> -> xs;
                     <2> z zs -> Cons{2,2} z (flipappend xs zs);
repeat x    = Cons{2,2} x (repeat x);
map f xs    = case xs of
                <1> -> Cons{1,0};
                <2> a b -> Cons{2,2} (f a) (map f b);
filter p xs = case xs of
                <1> -> Cons{1,0};
                <2> a b ->
   if (p a) (Cons{2,2} a (filter p b)) (filter p b);
zip as bs   = zipWith lam_zip as bs;
lam_zip a b = Cons{1,2} a b;
zipWith z xs ys =
 case xs of
   <1> -> Cons{1,0};
```

```
<2> a as -> case ys of
              <1> -> Cons{1,0};
              <2> b bs -> Cons{2,2}
                            (z a b)
                            (zipWith z as bs);
```

Literaturverzeichnis

[Bar84] BARENDREGT, H.P.: *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.

[BW88] BIRD, RICHARD und PHILIP WADLER: *Introduction to Functional Programming.* Prentice-Hall International, London, 1988.

[DM82] DAMAS, L. und R. MILNER: *Principal Type Schemes for Functional Programs.* In: *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, Seiten 207–212, Albuquerque, N.M., Januar 1982.

[DP90] DAVEY, B.A. und H.A. PRIESTLEY: *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, 1990.

[HAB+96] HAMMOND, K., L. AUGUSTSSON, B. BOUTEL, W. BURTON, J. FAIRBAIRN, J. FASEL, A. GORDON, M. GUZMÁN, J. HUGHES, P. HUDAK, T. JOHNSSON, M. JONES, D. KIEBURTZ, R. NIKHIL, W. PARTAIN, J. PETERSON, S. PEYTON JONES und P. WADLER: *Report on the programming language Haskell 1.3*, 1996.

[Jon95] JONES, MARK P.: *A system of constructor classes: overloading and implicit higher-order polymorphism.* Journal of Functional Programming, 5(1):1–37, 1995.

[Kic96] KICK, HUBERT: *Terminierungsanalyse für den Sprachkern einer nicht-strikten funktionalen Programmiersprache unter Verwendung eines Tableaukalküls für abstrakte Reduktion.* Diplomarbeit, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 1996.

[Man99] MANN, MATTHIAS: *Gleichheitsanalyse von Ausdrücken in nicht-strikten funktionalen Programmiersprachen unter Verwendung der Kontextanalyse.* Diplomarbeit, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 1999.

[Pan96] PANITZ, SVEN ERIC: *Termination Proofs for a Lazy Functional Language by Abstract Reduction.* Technischer Bericht Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt am Main, 1996. Verfügbar unter `www.ki.informatik.uni-frankfurt.de/papers/articles.html`.

[PJ87] PEYTON JONES, SIMON L.: *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[PJHA+99] PEYTON JONES, SIMON L., JOHN HUGHES, LENNART AUGUSTSSON, DAVE BARTON, BRIAN BOUTEL, WARREN BURTON, JOSEPH FASEL, KEVIN HAMMOND, RALF HINZE, PAUL HUDAK, THOMAS JOHNSSON, MARK JONES, JOHN LAUNCHBURY, ERIK MEIJER, JOHN PETERSON, ALASTAIR REID, COLIN RUNCIMAN und PHILIP WADLER: *Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language*, 1999.

[PJL91] PEYTON JONES, SIMON L. und DAVID R. LESTER: *Implementing Functional Languages: a Tutorial.* Prentice-Hall International, London, 1991.

[PvE93] PLASMEIJER, RINUS und MARKO VAN EEKELEN: *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Workingham, 1993.

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

[Sch99a] SCHÜTZ, MARKO: *ADE — Analysis of Demands (Arbeitstitel).* Implementierung von [Sch99b], in Arbeit, 1999.

[Sch99b] SCHÜTZ, MARKO: *Kontext-Analyse funktionaler Sprachen (Arbeitstitel).* Dissertation in Arbeit, 1999.

[SSPS95] SCHMIDT-SCHAUSS, M., S.E. PANITZ und M. SCHÜTZ: *Strictness Analysis by Abstract Reduction Using a Tableau Calculus.* In: MYCROFT, ALAN (Herausgeber): *Static Analysis Symposium '95*, Nummer 983 in *Lecture Notes in Computer Science*, Seiten 348–365. Springer, 1995.

[SSS97a] SCHÜTZ, MARKO und MANFRED SCHMIDT-SCHAUSS: *Automatic Extraction of Context Information from Programs Using Abstract Reduction.* Technischer Bericht Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt am Main, 1997. Verfügbar unter `www.ki.informatik.uni-frankfurt.de/papers/articles.html`.

[SSS97b] SCHÜTZ, MARKO und MANFRED SCHMIDT-SCHAUSS: *A Constructive Calculus Using Abstract Reduction for Context Analysis.* Unveröffentlicht, 1997.

[Wad95] WADLER, P.: *Monads for functional programming.* In: JEURING, J. und E. MEIJER (Herausgeber): *Advanced Functional Programming*, Nummer 925 in *Lecture Notes in Computer Science*, Seiten 24–52. Springer, 1995.

[Win93] WINSKEL, GLYNN: *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.