Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20)

Diplomarbeit

Gleichheitsanalyse von Ausdrücken in nicht-strikten funktionalen Programmiersprachen unter Verwendung der Kontextanalyse

Matthias Mann

11. Januar 1999

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

Danksagung

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

Matthias Mann

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

Frankfurt am Main, den 11. Januar 1999

Matthias Mann

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Es ist leicht abzusehen, daß die Komplexität von Softwaresystemen in der Zukunft weiterhin zunehmen wird. Im Bereich des Software-Engineering bzw. der Softwaretechnologie werden daher folgende Anforderungen an die Softwarequalität von besonderer Bedeutung sein:

- Zuverlässigkeit
- Wartbarkeit
- Erweiterbarkeit

Um diese Ziele zu erreichen, werden gemeinhin — auch besonders von den Vertretern des objektorientierten Paradigmas (siehe [Sch95], S.15ff.) — die Prinzipien *Abstraktion*, *Einkapselung*, *Hierarchiesierung* und *Modularisierung* als notwendig erachtet.

All diese Prinzipien werden von funktionalen Programmiersprachen bestens unterstützt. Darüber hinaus bieten diese gegenüber anderen bezüglich des Qualitätspunktes *Zuverlässigkeit* enorme Vorteile: Im Bereich der Programmverifikation fällt es wesentlich leichter, Aussagen über funktionale Programme zu beweisen, als das z.B. für imperative Sprachen mit ihrer um einiges komplizierteren Semantik möglich ist. Das ist dann erst recht der Fall, wenn es um das *automatische* Beweisen von Aussagen über Programme geht.

1.1.1 Anwendungen für eine Gleichheitsanalyse

Als Anwendungsbeispiel könnte man sich zwei Programme vorstellen, die dasselbe Ergebnis liefern sollen, dies aber auf unterschiedliche Weisen tun. Das erste Programm mag für den Menschen einfach zu verstehen sein, benötigt aber nachweislich übermäßig viel Rechenzeit. Das zweite kann einen schnelleren Algorithmus implementieren aber so kompliziert sein, daß unklar ist, ob es dasselbe Ergebnis wie das erste liefert. Ein Analysewerkzeug, welches Gleichheit von Programmen automatisch beweisen kann, ist hier sinnvoll einsetzbar.

Es kann aber auch als Teil eines Compilers dazu dienen, Programmtransformationen, die zu Optimierungszwecken angewendet werden, zu verifizieren bzw. die Grundlage für deren Anwendung zu liefern. Dabei hat die in dieser Arbeit behandelte Gleichheitsanalyse den zusätzlichen Vorteil, eine Beschreibung derjenigen Argumente zu liefern, für die zwei Funktionen gleich sind. Gilt die Gleichheit nämlich nicht auf sämtlichen Argumenten, so ist es sinnvoller, die positiven Ergebnisse aufzuzählen, anstatt die betreffenden Funktionen nicht als gleich zu beurteilen.

1.1.2 Grenzen der Gleichheitsanalyse

Denn die Gleichheitsanalyse kann ohnehin nur Antworten der Form „Beweis gefunden“ bzw. „Kein Beweis gefunden“ liefern. Der zweite Fall bedeutet dabei nicht, daß die Funktionen nicht gleich sind, sondern nur, daß eben keine Gleichheit nachgewiesen werden konnte.

Der Grund hierfür liegt in der Unentscheidbarkeit des Halteproblems. Nach dem *Satz von Rice* (vgl. dazu [HU79, Seite 185ff.]) ist daher auch die Gleichheit zweier Programme nicht entscheidbar.

1.2 Überblick

Es soll in dieser Arbeit also darum gehen, Gleichheit von Funktionen einer nicht-strikten — auch „verzögert auswertend“ genannten — funktionalen Programmiersprache zu untersuchen und maschinell zu beweisen.

In nicht-strikten Sprachen ist es im Gegensatz zu strikten möglich, potentiell unendliche Daten zu beschreiben. Das heißt, man kann zum Beispiel eine *allgemeine* Definition für die Liste *aller* Primzahlen angeben, die nur soweit ausgewertet wird, wie es jeweils nötig ist.

Damit wird sich die Vorgehensweise in dieser Arbeit deutlich von der bei strikten funktionalen Programmiersprachen unterscheiden. Denn dort werden Aussagen über Programme häufig mit Induktionsbeweisern verifiziert.

Auf unendlichen Listen z.B. ist aber eine Induktion über deren Länge bzw. Struktur nicht ohne weiteres durchführbar. In [BW88, Seite 169ff.] werden zwar Möglichkeiten erörtert, wie man diese strukturelle Induktion auch auf unendliche Listen erweitern kann. Es gibt aber Aussagen, die auch für das *take*-Lemma (vgl. [BW88, S. 182ff.]) nicht zugänglich sind, obwohl sie nicht außerordentlich schwierig erscheinen.

Daher wird in dieser Arbeit ein Kalkül entwickelt, der seine Aufgabe durch die Benutzung eines sog. Tableaus bewerkstelligt. Ein Tableau ist dabei im wesentlichen ein Baum, der einen Beweis für die an seiner Wurzel stehende Aussage darstellt.

Solch ein Beweis muß im Bezug auf den entsprechenden Begriff von „Gleichheit" korrekt sein. Diesen zu entwickeln, ist daher ein weiteres, zentrales Thema der Arbeit. Es wird sich dabei um eine Gleichheit handeln, die zwei Programme über ihr Verhalten bei der Einsetzung als Unterprogramm in ein weiteres Programm bzw. Programmfragment charakterisiert. Das führt notwendigerweise dazu, daß zwei gleiche Funktionen auch auf all ihren Argumenten gleich sein müssen.

Die Umkehrung dieser Eigenschaft heißt *Extensionalität.* Ist sie gegeben, so läßt sich die Gleichheit zweier Funktionen folgern, wenn sie sich auf allen Argumenten gleich verhalten. Wie wir später sehen werden, ist die Extensionalität des zugrundeliegenden Gleichheitsbegriffes für den Kalkül von entscheidender Wichtigkeit.

Aber zuerst werden in Kapitel 2 die wesentlichen Grundlagen für die Gleichheit von funktionalen Programmen erarbeitet. Dafür wird ein Blick auf die Λ-Sprache und verschiedene Gleichheitstheorien geworfen. Das umfaßt den $\boldsymbol{\lambda}$-Kalkül nach [Bar84] und den *lazy lambda calculus* aus [Abr90].

Mit Kapitel 3 schließt sich eine Einführung in das funktionale Programmieren an. Die Implementierung des Kalküls selbst erfolgt nämlich vollständig in der verzögert auswertenden, funktionalen Programmiersprache Haskell, und besonders wichtige Teile des Programmcodes werden, wann immer möglich, sogleich bei den Ideen vorgestellt.

Kapitel 4 beschäftigt sich mit der stark eingeschränkten aber turingmächtigen funktionalen Sprache, auf der die Analyse stattfindet. Hier wird auch die Gleichheit auf Ausdrücken dieser Kernsprache erklärt.

Damit kann schließlich in Kapitel 5 der Gleichheitskalkül und dessen Korrektheit behandelt werden. Außerdem wird dort die Implementierung des Kalküls ausführlich besprochen.

Anschließend werden in Kapitel 6 die mit dem Programm erzielten Ergebnisse dargestellt, bevor in Kapitel 7 eine Zusammenfassung mit Ausblicken auf mögliche Erweiterungen und weiterführende Untersuchungen die Arbeit abschließt.

Kapitel 2

Λ-Sprache und Gleichheitstheorien

In diesem Kapitel geht es um Theorien, die sich mit dem Gleichheitsbegriff auf λ-Termen beschäftigen. Dazu werden zuerst die Λ-Sprache selbst und einige wichtige Definitionen behandelt. Im darauf folgenden Abschnitt wird auf den „klassischen“ $\boldsymbol{\lambda}$-Kalkül nach [Bar84] eingegangen, welcher die theoretische Grundlage funktionaler Programmiersprachen darstellt. Die Betrachtungen dieses Kapitels werden schließlich mit dem „lazy“ $\boldsymbol{\lambda}$-Kalkül von Abramsky (siehe [Abr90]) abgerundet.

2.1 Die Syntax der Λ-Sprache

Definition 2.1.1. Wir definieren die Sprache Λ aller λ-Terme als $\Lambda = L(G)$ mittels der kontextfreien Grammatik

$$G = (\{M, x\}, V \cup \{\lambda, ., (,)\}, M, P)$$

wobei

$$P = \{M \to x,\ M \to (\lambda x.M),\ M \to (MM)\} \cup \{x \to v \mid v \in V\}$$

und V eine abzählbare Menge von *Variablen* bezeichne.

Wir fassen λ dabei in gewisser Weise als einen *Quantor*[1] auf und bezeichnen ein Vorkommen einer Variable x in einem Term $M \in \Lambda$ als *gebunden*, falls dieses Vorkommen von x innerhalb des (Geltungs-) Bereiches[2] eines λx auftaucht; ansonsten bezeichnen wir das Vorkommen von x als *frei*. Eine Variable kann durchaus frei *und* gebunden in einem Term vorkommen, z.B. x in $x(\lambda x.x)$.

[1] Wie z.B. $\forall$ und $\exists$ in der Prädikatenlogik.

[2] welcher bis zur zugehörigen schließenden Klammer reichen soll

Definition 2.1.2. Die Menge $\mathcal{FV}(M)$ der freien Variablen des Terms $M \in \Lambda$ wird definiert als die kleinste[3] Menge, die die folgenden Bedingungen erfüllt:

$$\begin{aligned} \mathcal{FV}(x) &= \{x\}, \text{ falls } x \text{ eine Variable ist.} \\ \mathcal{FV}(\lambda x.M) &= \mathcal{FV}(M) \setminus \{x\} \\ \mathcal{FV}(MN) &= \mathcal{FV}(M) \cup \mathcal{FV}(N) \end{aligned}$$

Als Menge der *geschlossenen* λ-Terme bezeichnen wir dann

$$\Lambda^0 = \{M \in \Lambda \mid \mathcal{FV}(M) = \emptyset\}$$

Ein geschlossener λ-Term heißt auch *Kombinator*. Einige Kombinatoren werden häufig benötigt und erhalten daher eigene Namen.

Definition 2.1.3. Wir definieren die folgenden Synonyme für geschlossene λ-Terme:

$$\begin{aligned} \boldsymbol{I} &\equiv \lambda x.x \\ \boldsymbol{K} &\equiv \lambda xy.x \\ \boldsymbol{S} &\equiv \lambda pqr.p\,r\,(qr) \\ \boldsymbol{Y} &\equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \\ \boldsymbol{\Omega} &\equiv (\lambda x.xx)(\lambda x.xx) \end{aligned}$$

Da wir in diesem Abschnitt bisher nur die Syntax der Λ-Sprache definiert haben, können wir Gleichheit zunächst einmal nur auf diese Weise ausdrücken; mit dem gerade benutzten Symbol $\equiv$ möchten wir diese syntaktische Gleichheit bezeichnen.

2.2 Wichtige Definitionen und Begriffe

Für den weiteren Umgang mit λ-Termen ist es sinnvoll, wie auch in [Bar84, Seite 26], eine Vereinbarung über die Benutzung von Variablennamen zu treffen. Dadurch wird eine Reihe von Betrachtungen wesentlich vereinfacht.

Konvention. Werden an einer Stelle in dieser Arbeit Aussagen über eine Menge von Termen gemacht, so wird stets angenommen, alle gebundenen Variablen seien so gewählt[4] worden, daß sie von den freien Variablen verschieden sind.

Damit kann recht einfach sichergestellt werden, daß eine Substitution die Struktur eines Termes respektiert, d.h. es wird vermieden, daß freie Variable nach der Anwendung der Substitution auf einmal im Bindungsbereich eines λ stehen.

[3] Diese Art von Definition wird im folgenden oft „induktiv“ genannt.

[4] Aus den Bemerkungen in Anschluß an Definition 2.3.1 wird deutlich werden, daß dies stets möglich und für die Bedeutung eines Termes unerheblich ist.

Definition 2.2.1. Sind $M, N \in \Lambda$ Terme und $x \in V$ eine Variable, so schreiben wir $M[N/x]$ für die Anwendung einer Substitution, die alle *freien* Vorkommen von x in M durch N ersetzt. Eine solche Substitution existiert und wird induktiv über die Struktur der λ-Terme definiert (siehe [Dav89, S. 154] bzw. [Bar84, S. 27]).

Beispiel 2.2.1. $(\lambda x.xy)[x/y]$ verstößt gegen die Konvention. Zulässig ist z.B. $(\lambda z.zy)[x/y]$, was korrekterweise $\lambda z.zx$ ergibt.

Im folgenden werden wir den Begriff *Term-* bzw. *Programmkontext* noch häufiger benötigen:

Definition 2.2.2. Ein *Termkontext*, kurz *Kontext*, wird induktiv definiert:

> $v \in V$ ist ein Kontext.
>
> $[\,]$ ist ein Kontext.
>
> Für beliebige Kontexte $C_1[\,]$ und $C_2[\,]$ sind $C_1[\,]\ C_2[\,]$ und $(\lambda x.C_1[\,])$ wieder Kontexte.

Die Notation $C[\ ] \in \Lambda$ soll bedeuten, daß $C[\ ]$ ein Kontext über Λ ist, d.h. $M \in \Lambda \implies C[M] \in \Lambda$.

Damit ist ein Kontext in gewisser Weise dual zu einem *Unter-* oder *Teilterm* zu sehen, der rekursiv definiert wird:

Definition 2.2.3. Sei $M \in \Lambda$ ein beliebiger Term, dann heißt $N \in \Lambda$ *Teilterm* von M, falls M eine der Formen

- $M \equiv N$,
- $M \equiv \lambda x.N$,
- $M \equiv M'N$ oder $M \equiv NM'$,

hat oder N ein Teilterm eines Teiltermes von M ist.

2.3 Der $\boldsymbol{\lambda}$-Kalkül

Der $\boldsymbol{\lambda}$-Kalkül formalisiert einen ersten Gleichheitsbegriff für λ-Terme. Wir definieren ihn nun ähnlich wie [Bar84] folgendermaßen:

Definition 2.3.1. Die formale Theorie $\boldsymbol{\lambda}$ besteht aus den Formeln $M = N$ für $M, N \in \Lambda$ und den folgenden Axiomen und Inferenzregeln:

1. Die *Konversionsregeln* bilden die Grundlage

$$(\alpha) \vdash\ (\lambda x.M) = (\lambda y.M[y/x])\ \ \forall M \in \Lambda,\ \text{falls } y \notin \mathcal{FV}(M).$$

$(\beta)\ \vdash\ (\lambda x.M)N = M[N/x]\ \ \forall M, N \in \Lambda.$

2. Die Gleichheit $=$ soll natürlich eine *Äquivalenzrelation* sein,

   $(\rho)\ \vdash\ M = M\ \ \forall M \in \Lambda.$

   $(\sigma)\ M = N\ \vdash\ N = M\ \ \forall M, N \in \Lambda.$

   $(\tau)\ M = N, N = L\ \vdash\ M = L\ \ \forall L, M, N \in \Lambda.$

3. und es soll *Ersetzbarkeit*[5] gelten

   $(\mu)\ N = N'\ \vdash\ MN = MN'\ \ \forall M, N, N' \in \Lambda.$

   $(\nu)\ M = M'\ \vdash\ MN = M'N\ \ \forall M, M', N \in \Lambda.$

   $(\xi)\ M = N\ \vdash\ (\lambda x.M) = (\lambda x.N)\ \ \forall M, N \in \Lambda.$

Existiert für zwei Terme $M, N \in \Lambda$ in $\boldsymbol{\lambda}$ ein Beweis der Formel $M = N$, so schreiben wir $\boldsymbol{\lambda}\ \vdash\ M = N$ — manchmal auch $M =_{\boldsymbol{\lambda}} N$ oder ganz kurz $M = N$ — und nennen M und N *konvertibel*. Gilt $M = N$ aufgrund der Regelmenge $\{\gamma_1, \ldots, \gamma_n\}$, so schreiben wir auch $M =_{\gamma_1,\ldots,\gamma_n} N$ und bezeichnen M und N als $\gamma_1, \ldots, \gamma_n$-äquivalent.

In der Definition 2.3.1 sind wir, bezogen auf die α-Konversion, [Dav89] gefolgt und haben sie im Gegensatz zu [Bar84] als eigene Inferenzregel aufgenommen. Ohne darauf ausdrücklich hinzuweisen, werden wir $\equiv$ trotzdem oft modulo $=_\alpha$ betrachten. Das heißt, Umbenennungen von gebundenen Variablen sollen keine Rolle spielen und die betreffenden Terme als *syntaktisch* gleich bezüglich $\equiv$ gelten.

Die gegebene Definition hingegen verdeutlicht, daß die Relation $=$ gerade als *Kongruenzabschluß* von $=_{\alpha,\beta}$ betrachtet werden kann.

2.3.1 Der Begriff der Kongruenz

Um dies näher zu beleuchten, muß zuerst einmal geklärt werden, was eine Kongruenz sein soll.

In [AU72, Seite 134] wird darunter ganz allgemein eine Relation verstanden, die links- und rechtsinvariant ist bezogen auf die Operationen in der Sprache. Diese Operationen sind in unserem Fall ja λ-Abstraktion und Applikation. Für letztere erkennt man in (μ) leicht die Links- und in (ν) die Rechtsinvarianz wieder.

Für die Λ-Sprache im speziellen definiert [Bar84, Seite 50] eine Kongruenz als eine Äquivalenzrelation auf Λ, die *kompatibel* zu den Operationen der Sprache ist.

Definition 2.3.2. Eine Relation $R \subseteq \Lambda \times \Lambda$ heißt *kompatibel* zu den Operationen der Λ-Sprache, falls aus $M\ R\ M'$ folgt

[5] engl. *substitutivity*

- $(\forall N \in \Lambda)\ MN\ R\ M'N$
- $(\forall N \in \Lambda)\ NM\ R\ NM'$
- $(\lambda\ x.M)\ R\ (\lambda\ x.M')$

Wie man sieht, entsprechen diese Anforderungen gerade wieder den Inferenzregeln (μ), (ν) und (ξ). Die Behauptung 3.2.1. aus [Bar84, Seite 59] rechtfertigt schließlich unsere Ansicht, von $=_{\boldsymbol{\lambda}}$ als Kongruenzabschluß der Relation $=_{\alpha,\beta}$ zu sprechen[6].

Nun ist es aber auch interessant, eine Kongruenz losgelöst von der konkreten Sprache — und somit deren Struktur — zu charakterisieren. Dies geschieht üblicherweise über Termkontexte:

Definition 2.3.3. Eine Äquivalenzrelation $R \subseteq L \times L$ ist eine *Kongruenz* gdw. $x\ R\ y \implies (\forall C[\,] \in L)\ C[x]\ R\ C[y]$ gilt. Erfüllt eine partielle Ordnung diese Eigenschaft, so nennen wir sie *Präkongruenz*.

Wie in [Bar84, Seite 50] wird eine Kongruenz in dieser Arbeit auch als *Gleichheit* bezeichnet. Eine Gleichheit sollte in der Tat folgende Eigenschaft erfüllen: Falls zwei Unterausdrücke als gleich gelten, dürfen sie in einem Term gegeneinander ausgetauscht werden, was gleichbedeutend mit $M = N \implies C[M] = C[N]$ ist.

Satz 2.3.1. *Die im $\boldsymbol{\lambda}$-Kalkül definierten Gleichungen[7] stellen eine Gleichheit in obigem Sinne dar: Für einen beliebigen Termkontext $C[\ ] \in \Lambda$ gilt $M =_{\boldsymbol{\lambda}} N \implies C[M] =_{\boldsymbol{\lambda}} C[N]$.*

Beweis. Induktiv über die Größe und Struktur der Termkontexte (siehe [Bar84, S. 29]). □

Somit hätten die letzten drei Inferenzregeln (μ), (ν) und (ξ) auch durch die einzige Regel

$$M = N\ \vdash\ C[M] = C[N]\ \ \forall M, N, C[\,] \in \Lambda$$

ausgedrückt werden können. Denn mit $C[\,] \equiv L\,[\,]$ folgt (μ), mit $C[\,] \equiv [\,]\,L$ dann (ν) und mit $C[\,] \equiv (\lambda x.[\,])$ schließlich (ξ). Die andere Richtung dieser Äquivalenz stellt Satz 2.3.1 dar.

2.3.2 Reduktion

Wie kann man sich nun die im $\boldsymbol{\lambda}$-Kalkül definierte Gleichheit vorstellen? Welche λ-Terme sind gleich? Einen operativen Eindruck davon kann man mit Hilfe der *β-Reduktion* gewinnen.

[6] Oder nur der Relation $=_\beta$ wie ebenda, wenn wir $\equiv$ modulo $=_\alpha$ sehen.

[7] Im Englischen würde man an dieser Stelle von *equations* sprechen, um nicht verwirrenderweise den Begriff „Gleichheit“ zweimal zu benutzen.

Definition 2.3.4. Ein Term der Gestalt $R \equiv (\lambda x.M)N$ heißt *β-Redex*[8] und $R' \equiv M[N/x]$ das dazugehörige *Kontraktum*. Für diesen Schritt vom Redex zum Kontraktum, den wir *β-Kontraktion* nennen, schreiben wir $R \to_\beta R'$.

Definition 2.3.5. Eine *β-Reduktion* ist nun eine Folge von mehreren β-Kontraktionen, und wird mit $\overset{*}{\to}_\beta$ notiert[9]. Das Ergebnis einer Reduktion bezeichnen wir als *Redukt*.

Eine β-Expansion ist das Entgegengesetzte zu einer Kontraktion, und eine Folge von mehreren Kontraktionen und/oder Expansionen nennen wir schließlich *β-Konversion*[10].

Definition 2.3.6. Enthält ein Term $M \in \Lambda$ keinen β-Redex der Form $(\lambda x.L)N$, so ist M eine *Normalform*, kurz NF.

Sieht man in einem naiven Versuch diese Normalform als Bedeutung von λ-Termen an, so identifiziert man gleichsam die Terme, die keine Normalform haben. Dies aber

- führt zu einer inkonsistenten[11] Theorie, wenn $\vdash \quad M = N$ für alle $M, N \in \Lambda$ ohne Normalform zu den Axiomen von $\boldsymbol{\lambda}$ hinzugefügt würde (siehe [Bar84, Seite 39]).

- steht im Widerspruch zum Modell der Theorie bzw. im Gegensatz zur Semantik der λ-Sprache: Haben M und N beide keine NF, so stellt dies nicht sicher, daß $[\![M]\!] = [\![N]\!]$, also M und N semantisch gleich sind (vgl. [Bar84, Seite 40]).

Vielmehr stellen sich als gleich im Modell all die Terme dar, die keine *Kopfnormalform* haben.

Definition 2.3.7. Ein Term $M \in \Lambda$ *ist* eine (oder „in") *Kopfnormalform*[12], kurz HNF, falls $M \equiv (\lambda x_1, \ldots, x_n.y\ N_1 \ldots N_n)$ und y eine Variable ist. Ein Term $M \in \lambda$ *hat* eine HNF, falls $\exists N\ M = N$ und N eine HNF ist.

Jetzt ist es konsistent, Terme ohne HNF als unlösbar bzw. undefiniert zu betrachten und somit gleichzusetzen. Denn sei $M \in \Lambda$ ein Term ohne HNF und N eine NF, dann gilt[13] für einen beliebigen Kontext $C[\,]$:

$$C[M] = N \implies (\forall L \in \Lambda)\ C[L] = N$$

In diesem Zusammenhang bezeichnet man Terme ohne HNF als *bedeutungslose* Terme.

---

[8] Abk. für *reducible expression*, engl. für „reduzierbarer Ausdruck".

[9] Formal gesehen handelt es sich dabei ja um die reflexiv-transitive Hülle von $\to_\beta$.

[10] entspricht diese doch gerade der β-Äquivalenz.

[11] D.h. alle Formeln $M = N$ mit geschlossenen Termen M und N können bewiesen werden.

[12] engl. *head normal form*

[13] siehe [Bar84, Seite 374].

2.3.3 Extensionalität

Man kann nun beobachten, daß sich alle Terme der Gestalt $\lambda x.Mx$ und M identisch *verhalten*. Das bedeutet, werden sie auf ein Argument angewandt, so können diese Applikationen ineinander überführt werden: $(\lambda x.Mx)N =_\beta MN$. Trotzdem können solche Äquivalenzen i.a. nicht bewiesen werden, z.B. $\boldsymbol{\lambda} \nvdash \lambda x.(\boldsymbol{\Omega} x) = \boldsymbol{\Omega}$, mit $\boldsymbol{\Omega}$ wie in Definition 2.1.3 angegeben. Würde

$$(\eta) \qquad \vdash\ (\lambda x.Mx) = M \quad \forall M \in \Lambda$$

als Axiom zur Theorie $\boldsymbol{\lambda}$ hinzugefügt werden, so hätte das den gleichen Effekt wie die folgende Inferenzregel:

$$(\text{ext}) \qquad Mx = Nx\ \vdash\ M = N \quad \forall M, N \in \Lambda, x \notin \mathcal{FV}(MN)$$

Wir könnten dann aus der Gleichheit als Funktion — d.h. die Anwendung auf Argumente liefert die gleichen Ergebnisse — die Gleichheit der Objekte folgern. Diese Eigenschaft heißt *Extensionalität.* Die um diese Regel erweiterte Theorie wird mit $\boldsymbol{\lambda} + \mathbf{ext}$ oder $\boldsymbol{\lambda\eta}$ bezeichnet. Beim Beweis[14] der Äquivalenz von $\boldsymbol{\lambda} + \mathbf{ext}$ und $\boldsymbol{\lambda\eta}$ spielt übrigens die Regel (ξ) eine entscheidende Rolle, weshalb sie auch *schwache Extensionalität* genannt wird.

2.4 Der „lazy“ $\boldsymbol{\lambda}$-Kalkül

Einen anderen Begriff von Gleichheit schlägt Abramsky in [Abr90] vor. Es handelt sich dabei um einen Ansatz, der Ausdrücke über ihr *Verhalten* bei der Einsetzung in einen Programmkontext vergleicht.

Seine Motivation für diese kontextuelle Gleichheit lag zwar eher in der Diskrepanz[15] zwischen modernen funktionalen Sprachen und der „Standardtheorie“, d.h. dem $\boldsymbol{\lambda}$-Kalkül, wie ihn Barendregt in [Bar84] behandelt und wir ihn in Abschnitt 2.3 kurz vorgestellt haben.

Der „lazy“ $\boldsymbol{\lambda}$-Kalkül bietet aber eine bedingte Form der Extensionalität, und die Grundprinzipien seiner Konstruktion sind auch für andere Begriffe von Gleichheit sehr gut geeignet.

Zwei geschlossene Terme $s, t \in \Lambda^0$ werden nämlich als gleich betrachtet, wenn es keinen Programmkontext $C[\,]$ gibt, so daß sich $C[s]$ „anders verhält“ als $C[t]$. Die Terme s und t sollen deshalb geschlossen sein, damit nicht freie Variablen in den Geltungsbereich von $C[\ ]$ geraten und die Beobachtung verfälschen. Das Verhalten, welches beobachtet werden soll, ist die Terminierung bzw. Nichtterminierung. Diese wird hier durch die sog. *schwache* Kopfnormalform charakterisiert:

[14] siehe [Bar84, Seite 32]

[15] Im $\boldsymbol{\lambda}$-Kalkül charakterisiert die HNF die Bedeutung von λ-Termen; in verzögert auswertenden funktionalen Sprachen wird aber unter einer λ-Abstraktion nicht weiter reduziert.

Definition 2.4.1. Ein Term $M \in \Lambda$ *ist* eine (oder „in“) *schwacher Kopfnormalform*[16], kurz WHNF, falls $M \equiv \lambda x.N$ ist. Analog zur Kopfnormalform sprechen wir dann davon, daß ein Term $M \in \Lambda$ eine WHNF *hat*, falls $\exists N \; M = N$ und N eine WHNF ist.

Im Sinne der Definition 2.3.5 kann man nun eine Relation „M reduziert zur schwachen Kopfnormalform N“ — in Zeichen $M \Downarrow N$ — folgendermaßen angeben:

Definition 2.4.2. Die Relation $\Downarrow \subseteq \Lambda^0 \times \Lambda^0$ wird induktiv definiert durch:

- Für $M \in \Lambda^0$ gelte $\lambda x.M \Downarrow \lambda x.M$.
- Falls $M \Downarrow \lambda x.P$ und $P[N/x] \Downarrow Q$ mit $M, N, P, Q \in \Lambda^0$, so gelte auch $MN \Downarrow Q$.

Wir schreiben kurz $M \Downarrow$ für $\exists N \; M \Downarrow N$ und $M \Uparrow$ für das Gegenteil.

2.4.1 Kontextuelle Ordnungsrelation

Die oben schon angedeutete Verhaltensgleichheit wird nun mit Hilfe einer kontextuellen Ordnungsrelation konstruiert. Diese Ordnungsrelation beschreibt im Prinzip, welche Ausdrücke seltener zu terminierenden Programmen führen. Dabei wird die Reduktion zu schwacher Kopfnormalform benutzt:

Definition 2.4.3. Die Relation $\leq_c \subseteq \Lambda^0 \times \Lambda^0$ wird definiert durch

$$M \leq_c N \iff \forall C[\,] \in \Lambda^0 \; C[M] \Downarrow \Longrightarrow C[N] \Downarrow$$

Man kann $\leq_c$ auf Λ erweitern, indem man freie Variablen gleichmäßig durch geschlossene Terme ersetzt, also

$$M \leq_c N \iff \forall \sigma : V \to \Lambda^0 \; M\sigma \leq_c N\sigma$$

Wann zwei Terme $M, N \in \Lambda$ jetzt als verhaltensgleich betrachtet werden können, ist klar:

$$M =_c N \iff M \leq_c N \wedge N \leq_c M$$

2.4.2 Bisimulation

Diese Definition ist aber noch recht unpraktisch, da wir, um die Gleichheit herauszufinden, nicht effektiv alle möglichen Kontexte durchprobieren können — es gibt ja unendlich viele. Daher ist es sinnvoll, eine äquivalente Definition zu finden, die ein schrittweises Vorgehen erlaubt. Mit einer Kette $\leq_{b,k}$ von Relationen über Λ^0 läßt sich dies bewerkstelligen:

[16] engl. *<u>w</u>eak <u>h</u>ead <u>n</u>ormal <u>f</u>orm*

Definition 2.4.4. Wir definieren $\leq_{b,k} \subseteq \Lambda^0 \times \Lambda^0$ induktiv

- Im Falle $k = 0$ sei $\leq_{b,0} = \Lambda^0 \times \Lambda^0$
- Für $k \geq 0$ gelte dann

$$M \leq_{b,k+1} N \iff M \Downarrow \lambda x.M_1 \implies (\exists N_1)\ N \Downarrow \lambda y.N_1 \wedge (\forall P \in \Lambda^0)\ M_1[P/x] \leq_{b,k} N_1[P/y]$$

Schließlich definiert man

$$M \leq_b N \iff \forall k \geq 0\ M \leq_{b,k} N$$

Man kann versuchen, diese Relation, die Abramksy als „applicative bisimulation" bezeichnet, in Worte zu fassen, wenn man sie sich dabei als Beobachtung eines mehrstufigen Experimentes vorstellt:

1. Am Anfang, bevor die erste Beobachtung gemacht wird, hat man noch keine Möglichkeit, zwei geschlossene Terme $M, N \in \Lambda^0$ zu differenzieren, daher gilt $M \leq_{b,0} N$ immer.

2. Auf der ersten Stufe des Experimentes können wir einen geschlossenen Term $M \in \Lambda^0$ auswerten und feststellen[17], ob M zu einer Kopfnormalform reduziert. Falls nicht, so gilt $M \leq_{b,1} N$ ohne Einschränkung, ansonsten nur, wenn auch N eine Kopfnormalform hat.

3. Für eine Kopfnormalform $\lambda x.M_1$ von M setzen wir die Beobachtung auf der nächsten Stufe fort, indem wir M_1 mit neuen Argumenten versorgen, d.h. wir versuchen nun $M[P/x]$ mit einem geschlossenen Term $P \in \Lambda^0$ zu WHNF zu reduzieren. Das weitere Vorgehen finge nun wieder bei 2 an, so daß wir

4. auf Stufe $k + 1$ angelangt, $M \leq_{b,k+1} N$ erhalten, wenn folgendes gilt:

   - Falls M eine WHNF $\lambda x.M_1$ hat, dann hat auch N eine WHNF, z.B. $\lambda y.N_1$ *und*
   - Für alle geschlossenen Terme $P \in \Lambda^0$ gilt $M_1[P/x] \leq_{b,k} N_1[P/y]$, d.h. wir versorgen die WHNF's mit allen möglichen geschlossenen Termen als Argumenten und für diese Anwendungen muß die Relation auf einer Ebene niedriger gelten.

[17] Dies ist zwar wiederum nicht effektiv, da wir *nicht entscheiden* können, ob die Auswertung zu WHNF terminiert, aber die Sichtweise ist stärker operational.

2.4.3 Ergebnisse

Abramsky zeigt dann die Übereinstimmung der Bisimulation mit der kontextuellen Gleichheit bzw. Ordnung als Spezialfall eines Satzes über „lambda transition systems“ ([Abr90, Proposition 6.6]):

Resultat. $\leq_c = \leq_b$.

Somit ist $\leq_b$ eine Präkongruenz bzw. $=_b$ eine Gleichheit und wir können eine Theorie definieren, die auf Ungleichungen bzw. Gleichungen basiert:

Definition 2.4.5. $\boldsymbol{\lambda} l = (\Lambda, \sqsubseteq, =)$ ist die formale Theorie mit den folgenden Inferenzregeln:

- $\boldsymbol{\lambda} l \vdash\ M \sqsubseteq N\ \ \forall M, N \in \Lambda$ mit $M \leq_b N$.
- $\boldsymbol{\lambda} l \vdash\ M = N\ \ \forall M, N \in \Lambda$ mit $M =_b N$.

Darüber hinaus gibt es noch weitere wichtige Ergebnisse, die im folgenden dargestellt werden sollen.

Resultat. Die Standardtheorie, d.h. der $\boldsymbol{\lambda}$-Kalkül aus Abschnitt 2.3, ist in der durch $=_b$ erzeugten $\boldsymbol{\lambda} l$-Theorie enthalten ([Abr90, Proposition 2.7]).

Damit ist gemeint, daß alle Formeln, die im $\boldsymbol{\lambda}$-Kalkül beweisbar sind, auch im $\boldsymbol{\lambda} l$-Kalkül bewiesen werden können.

$$\boldsymbol{\lambda} \vdash\ M = N \quad\Longrightarrow\quad \boldsymbol{\lambda} l \vdash\ M = N$$

Insbesondere ist die (β)-Inferenzregel in $\boldsymbol{\lambda} l$ herleitbar.

$$\boldsymbol{\lambda} l \vdash\ (\lambda\, x.M)N = M[N/x]$$

Von Wichtigkeit ist außerdem, daß die Nichtexistenz einer WHNF semantisch äquivalent ist zur Bedeutungslosigkeit eines Termes. Dies bezeichnet Abramsky als „Computational Adequacy“[18] (siehe [Abr90, Theorem 5.18]):

Resultat. Für alle $M \in \Lambda$ gilt

$$M \Downarrow \Longleftrightarrow\ [\![M]\!]^{D}_{\rho_\perp} \neq \perp$$

wobei $\perp$ für das kleinste Element im semantischen Bereich[19] D steht und $\rho_\perp$ alle Variablen auf $\perp$ abbildet.

Man kann zeigen, daß $\boldsymbol{\Omega}$ ein kleinstes Element bezüglich $\sqsubseteq$ ist. Dies ist nach dem obigen Resultat auch nicht weiter verwunderlich, da $\boldsymbol{\Omega}$ keine WHNF hat.

[18] Dieser Begriff wird in Abschnitt 3.1.3 erläutert.

[19] engl. *Domain*

Größte Elemente in λl

Es wirft aber die Frage auf, ob es ein größtes Element bezüglich $\sqsubseteq$ gibt. Dies ist der Fall, wie der folgende Satz zeigt (vgl. [Abr90, Proposition 2.7]).

Satz 2.4.1. *Der Term* $(\boldsymbol{Y}\ \boldsymbol{K})$ *ist ein größtes Element bezüglich* $\sqsubseteq$*, d.h. es gilt insbesondere für alle* $M \in \Lambda$

$$\boldsymbol{\lambda} l \vdash\ M \sqsubseteq (\boldsymbol{Y}\ \boldsymbol{K})$$

Beweis. Hierfür betrachten wir, wie $(\boldsymbol{Y}\ \boldsymbol{K})$ reduziert:

$$\begin{aligned}
(\boldsymbol{Y}\ \boldsymbol{K}) &\equiv (\lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx)))\ \ (\lambda yz.y) \\
&\rightarrow_\beta (\lambda x.(\lambda yz.y)(xx))\ (\lambda x.(\lambda yz.y)(xx)) \\
&\rightarrow_\beta (\lambda yz.y)\ ((\lambda x.(\lambda yz.y)(xx))(\lambda x.(\lambda yz.y)(xx))) \\
&\rightarrow_\beta (\lambda z.(\lambda x.(\lambda yz.y)(xx))\ (\lambda x.(\lambda yz.y)(xx))) \\
&\equiv \lambda z.\ \underbrace{(\lambda x.\boldsymbol{K}(xx))\ (\lambda x.\boldsymbol{K}(xx))}_{L}
\end{aligned}$$

Dabei ist $\lambda z.(\lambda x.\boldsymbol{K}(xx))(\lambda x.\boldsymbol{K}(xx))$ eine WHNF, und wir erhalten somit:

$$(\boldsymbol{Y}\ \boldsymbol{K}) \Downarrow \lambda z.L$$

Nun können wir zeigen, daß $M \leq_{b,k} (\boldsymbol{Y}\ \boldsymbol{K})$ für alle $M \in \Lambda$ und $k \in \mathbb{N}$ gilt. Dazu gehen wir für ein beliebiges $M \in \Lambda^0$ induktiv vor: Die Verankerung $M \leq_{b,0} (\boldsymbol{Y}\ \boldsymbol{K})$ gilt nach Definition von $\leq_{b,0}$ für jedes $M \in \Lambda^0$. Für $M \leq_{b,k+1} (\boldsymbol{Y}\ \boldsymbol{K})$ ist nichts zu zeigen, falls M keine WHNF hat. Gilt $M \Downarrow \lambda x.N$, so ist $\forall P \in \Lambda^0\ N[P/x] \leq_{b,k} L[P/z]$ zu prüfen, was aber wegen $L[P/z] \equiv (\lambda x.\boldsymbol{K}(xx))(\lambda x.\boldsymbol{K}(xx)) =_\beta (\boldsymbol{Y}\ \boldsymbol{K})$ gerade die Induktionshypothese darstellt. □

Wie oben schon bemerkt, gilt ja $M =_{\boldsymbol{\lambda}} N \implies M =_b N$. Dennoch definiert $\boldsymbol{\lambda} l$ nicht denselben Begriff von Gleichheit wie $\boldsymbol{\lambda}$, weil $=_\lambda \subsetneq =_b$ gilt, d.h. die Relation $=_b$ ist echt gröber als $=_\lambda$. Es gibt nämlich Terme, die zwar verhaltensgleich sind, aber *nicht* gleich bezüglich $=_{\boldsymbol{\lambda}}$.

Das einfachste Beispiel dafür ist ein weiterer größter Term in $\boldsymbol{\lambda} l$, nämlich $(\boldsymbol{Y2}\ \boldsymbol{K})$, wobei $\boldsymbol{Y2}$ definiert ist durch

$$\boldsymbol{Y2} \equiv (\lambda f.(\lambda xy.f(yxx))(\lambda xy.f(yxx))(\lambda xy.f(yxx)))$$

Wir zeigen analog zu $(\boldsymbol{Y}\ \boldsymbol{K})$, daß $(\boldsymbol{Y2}\ \boldsymbol{K})$ bezüglich $\sqsubseteq$ ein größtes Element ist, indem wir wieder die Reduktion betrachten:

$$\begin{aligned}
(\boldsymbol{Y2}\ \boldsymbol{K}) &\equiv (\lambda f.(\lambda xy.f(yxx))\ (\lambda xy.f(yxx))\ (\lambda xy.f(yxx)))\ \ (\lambda yz.y)\\
&\rightarrow_\beta (\lambda xy.(\lambda yz.y)(yxx))\ (\lambda xy.(\lambda yz.y)(yxx))\ (\lambda xy.(\lambda yz.y)(yxx))\\
&\rightarrow_\beta (\lambda yz.y)\ ((\lambda xy.(\lambda yz.y)(yxx))\\
&\qquad\qquad\quad (\lambda xy.(\lambda yz.y)(yxx))\\
&\qquad\qquad\quad (\lambda xy.(\lambda yz.y)(yxx)))\\
&\rightarrow_\beta (\lambda z.(\lambda xy.(\lambda yz.y)(yxx))\ (\lambda xy.(\lambda yz.y)(yxx))\ (\lambda xy.(\lambda yz.y)(yxx)))\\
&\equiv \lambda z.\ \underbrace{(\lambda xy.\boldsymbol{K}(yxx))\ (\lambda xy.\boldsymbol{K}(yxx))}_{L}
\end{aligned}$$

Da $\lambda z.(\lambda xy.\boldsymbol{K}(yxx))(\lambda xy.\boldsymbol{K}(yxx))$ eine WHNF ist, erhalten wir mit

$$(\boldsymbol{Y2}\ \boldsymbol{K}) \Downarrow \lambda z.L$$

dieselbe Grundlage für die Argumentation wie im Falle von $(\boldsymbol{Y}\ \boldsymbol{K})$. Nun wäre zu zeigen, daß $(\boldsymbol{Y}\ \boldsymbol{K})$ und $(\boldsymbol{Y2}\ \boldsymbol{K})$ nicht α,β-äquivalent sind. Hierfür genügte $\boldsymbol{Y} \neq_{\alpha,\beta} \boldsymbol{Y2}$, da $=_{\boldsymbol{\lambda}}$ eine Kongruenz ist.

Dies soll im folgenden aber nicht bewiesen, sondern nur anhand der Reduktion begründet werden. Man stellt dabei nämlich fest, daß $\boldsymbol{Y}$ keine Normalform hat. Vielmehr ergibt sich

$$\begin{aligned}
\boldsymbol{Y} &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\\
&\xrightarrow{*}_\beta (\lambda f.f((\lambda x.f(xx))(\lambda x.f(xx))))\\
&\xrightarrow{*}_\beta (\lambda f.f(f((\lambda x.f(xx))(\lambda x.f(xx)))))\\
&\xrightarrow{*}_\beta (\lambda f.f(f(\ldots f((\lambda x.f(xx))(\lambda x.f(xx))\ldots))))
\end{aligned}$$

Für $\boldsymbol{Y2}$ erhalten wir hingegen:

$$\begin{aligned}
\boldsymbol{Y2} &\equiv (\lambda f.(\lambda xy.f(yxx))(\lambda xy.f(yxx))(\lambda xy.f(yxx)))\\
&\xrightarrow{*}_\beta (\lambda f.f((\lambda xy.f(yxx))(\lambda xy.f(yxx))(\lambda xy.f(yxx))))\\
&\xrightarrow{*}_\beta (\lambda f.f(f((\lambda xy.f(yxx))(\lambda xy.f(yxx))(\lambda xy.f(yxx)))))\\
&\xrightarrow{*}_\beta (\lambda f.f(f(\ldots f((\lambda xy.f(yxx))(\lambda xy.f(yxx))(\lambda xy.f(yxx))\ldots))))
\end{aligned}$$

Wir sehen also, daß

$$\boldsymbol{Y} \xrightarrow{*}_\beta (\lambda f.f(f(f(\ldots f(M)))))$$

und

$$\boldsymbol{Y2} \xrightarrow{*}_\beta (\lambda f.f(f(f(\ldots f(N)))))$$

aber M und N weder aufeinander reduzierbar noch α-äquivalent sind.

Extensionalität

Wie auch im $\boldsymbol{\lambda}$-Kalkül kann $(\lambda x.\boldsymbol{\Omega}\ x) = \boldsymbol{\Omega}$ in $\boldsymbol{\lambda} l$ nicht bewiesen werden. Dagegen gibt es, wie in der Einleitung zu Abschnitt 2.4 schon angedeutet, eine bedingte Form der Extensionalität:

Resultat.

$$\boldsymbol{\lambda} l \vdash\ \lambda x.M\ x\ =\ M\ \ \forall M \Downarrow, x \notin \mathcal{FV}(M)$$

Dies bedeutet, daß $\boldsymbol{\lambda} l$ für alle Terme mit einer WHNF extensional ist. Wollte man volle Extensionalität erreichen, müßten nur noch die Problemfälle, d.h. die Terme ohne WHNF, behandelt werden. Das könnte man mit folgenden Definitionen erreichen.

Definition 2.4.6. Das Prädikat `XWHNF` sei gegeben durch

$$\texttt{XWHNF}(s) = s \neq_c \boldsymbol{\Omega} \wedge s \neq_c \lambda x_1 \ldots x_n.\boldsymbol{\Omega}\ \ \forall n > 0$$

Definition 2.4.7. Die kontextuelle Ordnung $\leq_{ce}$ wird nun durch

$$s \leq_{ce} t \iff \forall C[\ ]\ \texttt{XWHNF}(C[s]) \implies \texttt{XWHNF}(C[t])$$

definiert.

Dann folgt die Definition für die Gleichheit $=_{ce}$ und alles übrige analog zu $=_c$, bis darauf, daß $=_{ce}$ nun extensional ist (vgl. [SS97, Äquivalenzen, Seite 6f.]).

Kapitel 3

Grundlagen funktionalen Programmierens

Dieses Kapitel soll in Anlehnung an [FH88] bzw. [BW88] und [SS96] eine knappe Einführung in das funktionale Programmieren bieten.

Dabei sollen zuerst die Grundprinzipien des funktionalen Paradigmas im Vordergrund stehen und dessen Denkweise dem Leser nähergebracht werden. In diesem Zusammenhang werden auch einige Begriffe um die funktionale Programmierung erläutert. Der letzte Abschnitt des Kapitels stellt Haskell als einen Vertreter der funktionalen Sprachen kurz vor.

3.1 Das funktionale Paradigma

Das Wort „Paradigma“ stammt aus dem Griechischem und bedeutet in etwa soviel wie „Beispiel“, „Muster“ oder „Modell“. Dieser Begriff wird hier im Sinne von „Vorstellung, wie etwas funktioniert“ benutzt.

Beim funktionalen Programmieren geht die Vorstellung vom mathematischen Funktionsbegriff aus. Die wesentliche Operation ist die *Anwendung* einer Funktion auf ihre Argumente. Dazu ein Beispiel aus [FH88]:

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

Die Anwendung der Funktion max z.B. auf die Argumente 1 und 7 wird dann entsprechend der Definition *ausgewertet*:

$$max(1, 7) \rightarrow \left\{\begin{matrix} 1 & \text{falls } 1 > 7 \\ 7 & \text{sonst} \end{matrix}\right\} \rightarrow 7$$

Die Argumente einer Funktionsanwendung können ihrerseits natürlich wieder auf die gleiche Weise zusammengesetzt sein, also z.B.

$$max3(x, y, z) = max(x, max(y, z))$$

In der Mathematik müssen Funktionen wohldefiniert sein, das bedeutet, sie stellen eindeutige Abbildungen vom Definitions- in den Wertebereich dar. Dies stellt sicher, daß die Anwendung ein und derselben Funktion auf das gleiche Argument auch immer das gleiche Ergebnis liefert.

3.1.1 Referentielle Transparenz

Somit ist das Prinzip der *referentiellen Transparenz* gewährleistet. Das heißt, daß die Bedeutung eines Ausdrucks einzig und allein von seinen Teilausdrücken abhängt. Hierbei bringt die Auswertung einen Ausdruck nur in eine explizite Form, ändert aber nicht seinen Wert. Dadurch haben gleiche Teilausdrücke immer den gleichen Wert, auch wenn sie an verschiedenen Stellen im Programm stehen. Man könnte auch sagen, daß alle *Referenzen* auf einen Ausdruck genau seinem Wert entsprechen, was die Bezeichnung „referentielle Transparenz" erklärt.

Für die Realisierung der Gleichheitsanalyse spielt die referentielle Transparenz eine wichtige Rolle. Denn dadurch ist sowohl die Theorie als auch die Implementierung wesentlich einfacher zu handhaben, als wenn der Wert eines Ausdrucks von seiner Position im Programmtext abhinge.

In imperativen Programmiersprachen, z.B. PASCAL, ist dies hingegen der Fall, so daß wir die referentielle Transparenz dort u.U. verlieren können, wie das kleine Beispiel aus [FH88, Seite 11] zeigt:

Beispiel 3.1.1.

```
PROGRAM example;
VAR flag: BOOLEAN;

FUNCTION f(n: INTEGER): INTEGER;
BEGIN
  IF flag THEN
    f := n;
  ELSE
    f := 2 * n;
  flag := NOT flag;
END; (* f *)

BEGIN
  flag := TRUE;
  WriteLn(f(1) + f(2));
  WriteLn(f(2) + f(1));
END.
```

Weder ist der Wert des Ausdruckes `f(1)` an beiden Stellen im Programm der gleiche[1], noch ist die Kommutativität der Addition erfüllt. Durch die

[1] Für `f(2)` gilt dies natürlich analog.

Anweisung `flag := NOT flag;` wird nämlich eine *globale Variable geändert*. Es handelt sich um eine *destruktive Zuweisung*, die in die PASCAL-Funktion `f` einen *Seiteneffekt* einführt.

Seiteneffekte sind auch der wesentliche Grund dafür, daß sich die Gleichheitsanalyse, wie in der Einleitung der Arbeit schon angedeutet, für gewöhnliche imperative Sprachen als viel schwieriger erweist.

Funktionale Programmiersprachen, die keinerlei Seiteneffekte zulassen, nennt man *pur*. In puren funktionalen Sprachen ist referentielle Transparenz also stets ohne Einschränkungen gewährleistet.

3.1.2 Programmieren mit Funktionen

Wie schon erwähnt, bildet der $\boldsymbol{\lambda}$-Kalkül die theoretische Grundlage für die funktionale Programmierung. Daher hatten wir ja im vorigen Kapitel den dortigen Gleichheitsbegriff untersucht. Einmal abgesehen davon, ob man den $\boldsymbol{\lambda}$-Kalkül selbst überhaupt als eine Programmiersprache betrachten will (vgl. [Abr90, Seite 67]), möchte sicherlich niemand gerne auf diese Weise programmieren müssen. Wir haben uns in den vorangegangenen Abschnitten schon einige Betrachtungen erleichtern können, indem wir λ-Terme mit Namen versehen haben, wie zum Beispiel die Kombinatoren in Definition 2.1.3. Bis auf $\boldsymbol{\Omega}$ erfüllen all diese eine Eigenschaft, die nicht notwendigerweise nur auf geschlossene Terme zutreffen muß, sie sind nämlich „Lambda-geliftet“:

Definition 3.1.1. Sei $M \equiv \lambda\ x_1 \ldots x_n.t$ ein Term aus Λ, so daß gilt:

- $y \in \mathcal{FV}(t)$,
- $y \notin \{x_i \mid 1 \leq i \leq n\}$ und
- in t kommt kein λ vor

so kann M durch $((\lambda\ z\ x_1 \ldots x_n.t[z/y])\ y)$ ersetzt werden. Diesen Vorgang nennt man *Lambda-Lifting*.

Für geschlossene Terme, die Lambda-geliftet sind, gibt es eine spezielle Bezeichnung:

Definition 3.1.2. Ist ein Term $t \in \Lambda^0$ Lambda-geliftet, so heißt er *Superkombinator*.

Superkombinatoren kann man mit einem Namen versehen und als definierte Funktionssymbole betrachten. Man hätte den Superkombinator $\boldsymbol{K}$ also auch ohne weiteres durch $\boldsymbol{K}\ x\ y\ =\ x$ definieren können.

Genau das ist in den meisten funktionalen Programmiersprachen möglich, nämlich Superkombinatoren über Gleichungen zu definieren. Diese Gleichungen dürfen rekursiv sein, d.h. die rechte Seite darf nicht nur Namen anderer definierter Superkombinatoren sondern auch den Namen des zu definierenden Superkombinators selbst enthalten. Ein Programm besteht dann aus einer Reihe solcher Superkombinatordefinitionen und einem ausgezeichneten Ausdruck, den es bei Ablauf des Programmes auszuwerten gilt.

Dadurch, daß sich mit Hilfe des Lambda-Lifting jeder λ-Term in eine Anwendung eines Superkombinators übersetzen läßt (siehe dazu [PJ87] bzw. [PvE93]), erhalten wir folgendes

Resultat. Jede Programmiersprache, in der die Definition von Superkombinatoren möglich ist, ist turingmächtig[2].

3.1.3 Bedeutung von Ausdrücken

Nun ist aber nicht aus jeder auf diese Art angegebenen Superkombinatordefinition sofort ersichtlich, welche Funktion damit beschrieben werden soll. Durch eine Gleichung wie $\boldsymbol{nt} = \boldsymbol{nt}$ ist nicht direkt eine wohldefinierte mathematische Funktion erklärt.

Das müssen wir erst noch tun, indem jedem Superkombinator ein semantischer Wert zugewiesen wird, der dann eine wohldefinierte mathematische Funktion darstellt. Dieser Wert ist die *Bedeutung* des Superkombinators, und wir sprechen auch davon, daß jeder Ausdruck einen Wert *bezeichnet*. Hiervon rührt der Name *denotational*[3], den diese Art von Semantik trägt. Später werden wir die Begriffe Superkombinator und Funktion nicht mehr immer so klar trennen, daß Superkombinator für die textuelle Definition und Funktion für dessen semantischen Wert steht.

Im vorigen Abschnitt 3.1.1 wurde ja schon im Bezug auf die referentielle Transparenz postuliert, daß die Auswertung eines Ausdruckes dessen Wert nicht verändern darf.

Nun haben wir auch ein Mittel zur Hand, mit dem wir dies sicherstellen können. Haben wir erst einmal eine denotationale Semantik für eine Sprache angegeben, und damit die Bedeutung von Ausdrücken festgelegt, so können wir überprüfen, daß sich diese während der Auswertung nicht verändert. Ist dies wirklich der Fall, so spricht man von der *Adäquatheit* der Auswertung[4].

3.1.4 Auswertungsreihenfolge

Die Adäquatheit der Auswertung ist nicht offensichtlich, gibt es doch verschiedene Strategien, einen Ausdruck auszuwerten. Im folgenden werden die beiden wichtigsten Strategien, die zugleich gegensätzliche Extreme darstellen, kurz behandelt. Darüber hinaus sind zwischen den beiden endlos viele Mischstrategien denkbar.

[2] Von Beschränkungen, die z.B. ein Typsystem mit sich bringen kann, sei hier einmal abgesehen.

[3] engl. *to denote* – bezeichnen, bedeuten

[4] engl. *computational adequacy*

Anwendungsordnung

Die eine heißt *Applikations-* oder *Anwendungsordnung* und ihr Vorgehen besteht darin, bei einer Applikation $\boldsymbol{f}\ t$ zuerst den Argumentterm t auszuwerten und das Ergebnis dann in die Definitionsgleichung des Superkombinators $\boldsymbol{f}$ einzusetzen.

Normalordnung

Die Normalordnung verfolgt genau das entgegengesetzte Ziel, indem bei einer Applikation $\boldsymbol{f}\ t$ das Argument t immer sofort in die Definitionsgleichung von $\boldsymbol{f}$ eingesetzt wird. Der Argumentterm t wird dabei in den folgenden Schritten nur ausgewertet, wenn es die Auswertung auch wirklich verlangt.

Um den Unterschied zu verdeutlichen, betrachten wir den Ausdruck $\boldsymbol{K}\ s\ t$, wobei wir uns t als einen Term vorstellen, dessen Auswertung viele Ressourcen — Zeit wie Platz — verschlingen möge.

In der Anwendungsordnung wird nun zuerst dieser Aufwand getrieben, um dann die berechneten Ergebnisse von s und t in die Gleichung für $\boldsymbol{K}$ einzusetzen und schließlich s zu erhalten. Dadurch, daß dies bei der Normalordnung sofort geschieht, ersparen wir uns die Berechnung von t und berechnen einzig s.

Terminierung

Offensichtlicher wird dies noch, wenn es sich bei t um einen Term handelt, dessen Auswertung nicht terminiert, also z.B. $\boldsymbol{nt}$ definiert durch $\boldsymbol{nt} = \boldsymbol{nt}$. Dann nämlich terminiert auch die Auswertung von $\boldsymbol{K}\ s\ \boldsymbol{nt}$ in Anwendungsordnung nicht, wohl aber die in Normalordnung.

Allgemein ist es so, daß die Auswertung in Normalordnung immer dann terminiert, falls es überhaupt eine Auswertung gibt, die terminiert. In keinem Fall terminiert aber die Auswertung mittels einer Strategie mit einem anderen Ergebnis als die mittels einer anderen Strategie (vgl. [SS96]).

Striktheit

Funktionale Programmiersprachen, die in Anwendungsordnung ausgewertet werden, nennt man *strikt*. Solche Sprachen sind z.B. Scheme (als Vertreter der Lisp-Familie, siehe [AS85]) und ML ([MTH90]).

Diejenigen funktionalen Sprachen, die eine zur Auswertung in Normalordnung adäquate Semantik haben, heißen *nicht-strikt*, *verzögert auswertend* oder auch *lazy*[5]. Hierzu sind besonders Miranda ([Tur85]), Clean (vgl. [PvE93]) und Haskell ([PHA+97]) zu nennen.

In den Implementierungen dieser Sprachen mußte Sorge dafür getragen werden, daß gleiche Ausdrücke nicht mehrfach berechnet werden. Dies kann

[5]engl. *lazy* – faul

nämlich in der normalen Auswertungsreihenfolge geschehen, wenn die Terme durch Bäume darstellt sind und in der Definition einer Funktion ein Argument mehrmals auftritt. Man betrachte dazu folgendes Beispiel

$$\textbf{\textit{double}}\ x \;=\; x \;+\; x$$

bei dem angenommen wird, daß + als eine primitive Funktion zur Verfügung steht. Möchten wir nun ***double*** (3+4) berechnen, so besteht der erste Schritt in Normalordnung aus

$$\textbf{\textit{double}}\ (3+4) \;\rightarrow\; (3+4) \;+\; (3+4)$$

Haben wir uns nicht „gemerkt", daß der Term $(3 + 4)$ jeweils der gleiche ist, was gerade durch das zweimalige Auftauchen von x in der rechten Seite der Definition von ***double*** hervorgerufen wurde, so würden wir ihn zweimal berechnen.

Sharing

Um das zu vermeiden, werden mittlerweile in allen Implementierungen nichtstrikter funktionaler Programmiersprachen Graphen anstelle von Bäumen verwendet. Dabei wird ein mehrfaches Auftreten eines Terms einfach durch mehrere Referenzen auf ein und denselben Knoten im Graphen dargestellt. In dieser Repräsentation wird der Inhalt eines Knoten nach dessen Auswertung mit dem Ergebnis überschrieben.

Dadurch steht es sofort auch anderen Knoten zur Verfügung, die diesen referenzieren[6], d.h. das Ergebnis wird von all diesen Knoten gemeinsam benutzt, weswegen die Technik *Sharing*[7] genannt wird.

Sie führt dazu, daß bei der Auswertung in Normalordnung *höchstens* soviele Schritte durchgeführt werden müssen wie in Anwendungsordnung.

3.2 Haskell: Eine nicht-strikte, pure funktionale Sprache

Von den genannten Programmiersprachen legen wir unser Augenmerk nun auf Haskell, da die Implementierung der Gleichheitsanalyse selbst in dieser nicht-strikten, puren funktionalen Sprache erfolgt.

Ein kleines Beispielprogramm soll einen Einstieg in die Syntax von Haskell geben. Außerdem werden dabei einige Aspekte des Typsystems angesprochen. Selbstverständlich können die Ausführungen hier die vollständige Referenz in [PHA+97] nicht ersetzen.

[6] Man sieht schnell, daß dieses Vorgehen referentielle Transparenz voraussetzt und daher nur in einer puren funktionalen Sprache möglich ist.

[7] engl. *to share* – teilen, gemeinsam haben

Haskell erlaubt sogenannte *literate scripts*, d.h. Quellcodedateien, in denen die Beziehung zwischen Programmcode und Kommentar auf den Kopf gestellt wird. Nur noch Zeilen, die mit einem > beginnen, gehören zum Programm, der Rest ist Kommentar. Genau das macht sich diese Arbeit zunutze, indem das Dokument gleichzeitig auch das ausführbare bzw. übersetzbare Analyseprogramm darstellt.

3.2.1 Module

Das folgende Modul ist für das Funktionieren der Analyse natürlich nicht notwendig.

```
> module DemoProgramm (WieListe(..),
>                      Anwenden,
>                      istErstesGerade) where
> import List
```

Wie man sieht, wird ein Modul eingeleitet durch das Schlüsselwort `module` gefolgt von einer Liste der exportierten Bezeichner in Klammern und einer Liste von `import`-Anweisungen, mit denen Funktionen und Datentypen aus anderen Modulen bereitgestellt werden.

3.2.2 Algebraische Datentypen

In Haskell können *algebraische Datentypen* durch die Angabe ihrer *Konstruktoren* definiert werden:

```
> data WieListe a = Leer
>                 | MitWas a (WieListe a)
```

Der Datentyp `WieListe` ist — wie eine Liste eben — entweder leer, oder besteht aus irgendeinem Element vom Typ `a` und einer Restliste. Wie wir sehen, handelt es sich um einen *polymorphen* Typen, da wir den Elementtyp noch nicht festlegen müssen. Bei diesem kann es sich später um Zahlen, Wahrheitswerte oder wieder Elemente vom Typ `WieListe` handeln. Die Angabe von `WieListe(..)` in der Exportliste exportiert nicht nur den *Typkonstruktor* `WieListe` sondern auch dessen *Datenkonstruktoren* `Leer` und `MitWas`.

3.2.3 Pattern Matching

Möchte man wissen, mit welchem Konstruktor ein Datenobjekt gebildet wurde, kommt das *Pattern Matching* ins Spiel. Dies wird in der Funktion `wendeAnAufWieListe`, die eine Funktion `f` auf jedes Element einer `WieListe` anwendet, benutzt:

```
> wendeAnAufWieListe f Leer = Leer

> wendeAnAufWieListe f (MitWas was rest) =
>   MitWas (f was) (wendeAnAufWieListe f rest)
```

Die Funktion `wendeAnAufWieListe` wird also einfach für jeden Datenkonstruktor separat definiert.

3.2.4 Typklassen

Möchte man bestimmte Funktionen auf einer ganzen Menge von Datentypen zur Verfügung stellen, so bildet man sinnvollerweise eine *Typklasse*.

```
> class Anwenden c where
>   wendeAn :: (a -> b) -> c a -> c b
```

Das bedeutet, daß es in der Klasse `Anwenden` eine *Methode* gibt. Diese ist `wendeAn`, deren Typ nach den `::` angegeben ist. Es handelt sich um eine Funktion, die als erstes Argument eine Funktion von `a` nach `b` bekommt und als nächtes ein Element vom Typ der Klasse `c`, in dem ein Element vom Typ `a` enthalten ist. Das Ergebnis ist dann ein Element vom Typ der Klasse `c`, das nun ein Element vom Typ `b` enthält.

Genau das macht `wendeAnAufWieListe`, weswegen wir `WieListe` als eine *Instanz* von `Anwenden` einrichten können:

```
> instance Anwenden WieListe where
>   wendeAn = wendeAnAufWieListe
```

Im Prinzip war dieses Beispiel nichts anderes als die Listeninstanz für die Typklasse `Functor` (vgl. [PHA+97, S. 88 bzw. 94]).

3.2.5 Guards

Ein weiteres, häufig verwendetes, Sprachelement soll noch besprochen werden. *Guards* fangen mit einem `|` an und dienen dazu, Funktionen bedingt zu definieren, wie das folgende Beispiel zeigt:

```
> istErstesGerade Leer = "Nein, leer!"
> istErstesGerade (MitWas erstes _)
>    | even erstes = "Ja, das ist es."
>    | otherwise   = "Nein, es ist ungerade."
```

Falls das Prädikat `even erstes` zutrifft, so wird `"Ja, das ist es."` zurückgegeben, ansonsten der Wert hinter `otherwise`, welches nur ein Synonym für den Wahrheitswert `True` darstellt. Außerdem wurde in diesem Beispiel der Unterstrich `_` benutzt, der dem Compiler bei dem Pattern Match auf `MitWas` mitteilt, daß die zweite Komponente, also die Restliste, uninteressant ist.

Kapitel 4

Eine funktionale Kernsprache mit algebraischen Datentypen

In diesem Kapitel wird zuerst die Syntax einer funktionalen Kernsprache definiert. Deren Struktur ist zwar recht einfach gehalten, um die Gleichheitsanalyse später nicht zu kompliziert werden zu lassen. Sie ist aber mächtig genug, jedes funktionale Programm auszudrücken. Wir erklären dann die Semantik der Kernsprache operational, und der letzte aber wohl wichtigste Abschnitt befaßt sich mit dem Gleichheitsbegriff in dieser Sprache.

4.1 Die Syntax der Kernsprache

Die Kernsprache lehnt sich an die aus [PJL91] an, wird aber noch vereinfacht, indem wir λ-Ausdrücke und `let` fortlassen. Mittels Lambda-Lifting lassen sich nämlich auch diese durch Superkombinatoren ausdrücken (vgl. [PJ87] bzw. [PvE93]). Die Syntax der funktionalen Kernsprache Λ_C mit algebraischen Datentypen wird durch eine kontextfreie Grammatik spezifiziert. Deren Produktionenmenge P ist in Abbildung 4.1 zu sehen. Welche Symbole jeweils die Menge T der Terminale und die Menge N der Nichtterminale bilden, ist aus der typographischen Darstellung dort erkennbar.

Definition 4.1.1. Die Sprache Λ_C stellt die Menge aller gültigen Programme dar und wird durch $\Lambda_C = L(G)$ mittels $G = (N, T, P, prog)$ definiert.

Ein Programm besteht also aus einer Reihe von Superkombinatordefinitionen. Um es ablaufen zu lassen, benötigt man einen ausgezeichneten Ausdruck, mit dem die Auswertung beginnen soll, üblicherweise `main`.

Im Gegensatz zum $\boldsymbol{\lambda}$-Kalkül hängt die Bedeutung eines Ausdruckes wiederum vom Programm ab, in dem die verwendeten Superkombinatoren definiert sind. Dies zeigt, daß die Begriffe Programm und Ausdruck nicht voll-

Programm	*prog*	$\rightarrow$	sk_1 ; ... ; sk_n [;]	$n \geq 1$
Super-kombinator	*sk*	$\rightarrow$	$var\ var_1 \ldots var_n$ `=` *expr*	$n \geq 0$
Ausdruck	*expr*	$\rightarrow$	*expr aexpr*	Anwendung
		|	$expr_1$ *binop* $expr_2$	Infix binäre Anwendung
		|	`case` *expr* `of` *alts*	Case Ausdruck
		|	*aexpr*	Atomarer Ausdruck
Atomarer Ausdruck	*aexpr*	$\rightarrow$	*var*	Variable
		|	*num*	Zahl
		|	`Cons{`*num*`,`*num*`}`	Konstruktor
		|	`(` *expr* `)`	Geklammerter Ausdruck
Alternativen	*alts*	$\rightarrow$	alt_1 ; ... ; alt_n	$n \geq 1$
	alt	$\rightarrow$	`<`*num*`>` $var_1 \ldots var_n$ `->` *expr*	$n \geq 0$
Binärer Operator	*binop*	$\rightarrow$	*arop* | *relop* | *boolop*	
	arop	$\rightarrow$	`+` | `-` | `*` | `/`	Arithmetik
	relop	$\rightarrow$	`<` | `<=` | `==` | `~=` | `>=` | `>`	Vergleich
	boolop	$\rightarrow$	`&` | `|`	Bool'sch
Variable	*var*	$\rightarrow$	*alpha* $varch_1 \ldots varch_n$	$n \geq 0$
Zahl	*num*	$\rightarrow$	$ziffer_1 \ldots ziffer_n$	$n \geq 1$
	alpha	$\rightarrow$	*ein alphabetisches Zeichen*	
		|	`_`	Unterstrich
	varch	$\rightarrow$	*alpha* | *ziffer*	
	digit	$\rightarrow$	*ein numerisches Zeichen*	

Abbildung 4.1: Die Syntax der Kernsprache Λ_C

kommen voneinander getrennt werden können und folgende Definition berechtigt ist.

Definition 4.1.2. Die Menge aller *Ausdrücke* in der Kernsprache wird durch die Grammatik $E = (N, T, P, expr)$ repräsentiert und ebenfalls mit Λ_C bezeichnet.

Wir schreiben also Λ_C sowohl für die Menge aller Kernsprachenprogramme als auch -ausdrücke. Aus dem jeweiligen Kontext geht hervor, in welchem speziellen Sinne Λ_C verwendet wird.

Das Symbol $\equiv$ drückt wie in Abschnitt 2.1 die syntaktische Äquivalenz von Termen aus. Auch die Notation $[e/x]$ für die Substitution wird aus 2.2 übernommen.

4.1.1 Algebraische Datentypen

Ein formales Typsystem für Λ_C wird nicht eingeführt. Denn wir gehen davon aus, daß wir stets, z.B. als Zwischenprodukt eines Compilers, ein wohlgetyptes Programm erhalten. In Abschnitt 4.4.1 werden wir genau erklären, was wir dabei unter wohlgetypt verstehen.

Mit den Konstruktorausdrücken $\texttt{Cons}\{j, m\}$ können Datenobjekte konstruiert werden, wobei j ein Bezeichner für den Konstruktor und m dessen Stelligkeit, d.h. die Anzahl seiner Argumente, ist. Diese Datenobjekte können dann über `case` unterschieden werden, welches in seinen Alternativen entsprechend des Bezeichners verzweigt. Dazu sind in Abschnitt 4.2, in dem die Auswertung diskutiert wird, detailliertere Informationen zu finden.

In der Syntax der Kernsprache gibt es — im Gegensatz zu Haskell beispielsweise — keine Unterscheidungsmöglichkeit mehr zwischen *verschiedenen* algebraischen Datentypen.

Es wird aber vorausgesetzt, daß es in allen Kernsprachenprogrammen eine Menge $\mathcal{A}$ von algebraischen Datentypen $A_1, \ldots, A_n$ gibt. Jedem solchen Typen A wird jeweils eine Anzahl $|A|$ von Konstruktoren zugeordnet, die mit $\texttt{con}_{A,1}, \ldots, \texttt{con}_{A,|A|}$ bezeichnet wird. Wie oben schon bemerkt, hat jeder Konstruktor $\texttt{con}_{A,i}$ eine Stelligkeit m, für die wir oft $ar(\texttt{con}_{A,i})$ notieren, d.h. $ar(\texttt{con}_{A,i_j}) = m \iff \texttt{con}_{A,i_j} \equiv \texttt{Cons}\{j, m\}$.

Beispiel 4.1.1. In folgendem Kernsprachenprogramm gibt es einen algebraischen Datentypen, nennen wir ihn *List*, zu dem die zwei Konstruktoren `Cons{1,0}` und `Cons{2,2}` gehören.

```
map f as = case as of
  <1>      -> Cons{1,0};
  <2> x xs -> Cons{2,2} (f x) (map f xs);
```

`map` wendet die Funktion `f` auf jedes Element der Liste `as` an, siehe dazu auch Abschnitt 3.2.3.

4.1.2 Ein Parser für Λ_C

Für die Gleichheitsanalyse bedeutet das Kernsprachenprogramm einen wesentlichen Teil der Eingabe, weshalb auch gleich ein Parser für Λ_C beschrieben werden soll. Dies geschieht in Form einer Grammatikdatei für den Parsergenerator `lucky`, den Norbert Klose entwickelt und in [Klo97] beschrieben hat. Die Grammatikdatei hat im wesentlichen die gleiche Struktur wie die des `lucky`-Vorbildes `happy` (vgl. [GM96]).

Zu Beginn kann man, in geschweifte Klammern `{` und `}` eingeschlossen, einen Vorspann angeben, der unverändert in das Haskell-Programm übernommen wird. Wir benötigen an dieser Stelle sowohl die Moduldefinition

```
> {

> module Parser(lexit, syntax, parse, Lexem, Token) where
> import Datadefs
> import Utils
> import Char
```

als auch einen Datentypen für die Lexeme, in die der Lexer den Programmtext zerlegt. Lexeme sind alle Schlüsselworte von Λ_C, also z.B.

```
> data Lexem
>   = LexemCase             -- 'case'
>   | LexemOf               -- 'of'
```

oder

Zahlen, dargestellt als `Int`.

Variablen mit ihrem Namen als `String`.

Zeichen, die ursprünglich in ' eingeschlossen waren und als `String` gespeichert werden. So können auch Sonderzeichen wie `\n` usf. dargestellt werden.

Zeichenketten, die ursprünglich in " eingeschlossen waren und ebenfalls als `String` dargestellt werden.

```
>   | LexemNum Int          -- number
>   | LexemVar String       -- variable
>   | LexemChar String      -- char
>   | LexemStr String       -- string
```

Darüber hinaus gibt es noch einige Lexeme, die aus einem oder zwei Zeichen bestehen, hier aber nicht aufgeführt werden sollen. Es handelt sich bei diesen um sämtlich Operatorsymbole sowie das `=` für Superkombinatordefinitionen.

Der Typ `Token` beinhaltet nun sowohl ein solches `Lexem` als auch dessen Position (`Line` und `Col`) im Quelltext.

```
> type Line = Int
> type Col = Int
> type Token = (Lexem, Line, Col)
```

Damit kann eine syntaktisch fehlerhafte Stelle im Quelltext recht genau angegeben werden.

```
> }
```

In der Beschreibungsdatei für den Parser folgt dann ein Teil mit Direktiven, um dem Parser einen Namen zu geben, den Tokentyp und die Token selbst festzulegen.

```
> %name syntax
> %tokentype { Token }

> %token
>   case     { (LexemCase, _, _) }
>   of       { (LexemOf, _, _) }
>   num      { (LexemNum $$, _, _) }
>   var      { (LexemVar $$, _, _) }
>   char     { (LexemChar $$, _, _) }
>   str      { (LexemStr $$, _, _) }
```

Das `$$` hat dabei die Bedeutung, daß der *Wert* eines Tokens, z.B. `num`, nicht das Lexem selbst, sondern in diesem Fall die entsprechende Zahl sein soll. Der Wert des Tokens `case` z.B. ist nämlich einfach `LexemCase` und auch die Token für die übrigen Lexeme werden analog definiert.

Durch ein doppeltes `%` von den Direktiven getrennt, beginnt dann der Teil mit der Grammatik für die Kernsprache. Dabei stellt `CoreExpr` den üblichen algebraischen Datentypen für die Kernsprache dar. Für die Implementierung dieser Arbeit konnten Teile der Terminierungsanalyse von Hubert Kick (siehe [Kic96]) weiterverwendet werden. Daher werden u.a. auch die Datentypen `CoreExpr`, `CoreProgram` und `CoreScDefn` von dort übernommen. Deren Definitionen stimmen im wesentlichen mit denen aus [PJL91, Seite 8ff.] überein.

Ein Programm besteht jetzt aus Definitionen von Superkombinatoren, die mit einem Semikolon abgeschlossen werden.

```
> %%

> Program :: {CoreProgram}
>    : Sc ';' Program   { $1 : $3 }
>    | Sc ';'           { [$1] }

> Sc :: {CoreScDefn}
>    : var '=' Expr                 { ($1, [], $3) }
>    | var VarNamesList '=' Expr    { ($1, $2, $4) }
```

Ein Superkombinator kann als sogenannte CAF[1] eine leere Parameterliste haben, oder nach dem Namen des Superkombinators folgt eine Liste von Variablennamen, die mindestens ein Element enthält.

```
> VarNamesList :: {[Name]}
>    : var                 { [$1] }
>    | var VarNamesList    { $1 : $2 }
```

Der Parsergenerator `lucky` unterstützt nicht wie `yacc` die Definition von Operator-Präzedenzen. Um Boolsche und arithmetische Ausdrücke mit der korrekten Rangfolge der Operatoren angeben zu können, muß die Grammatik daher in eine entsprechende Form gebracht werden[2]. Da man diese auch in [PJL91, Seite 30] findet, sei hiervon nur der Anfang wiedergegeben.

```
> Expr :: {CoreExpr}
>    : case Expr of Alts            { ECase $2 $4 }
>    | Expr1                        { $1 }
```

Ein Ausdruck ist entweder ein `case` oder ein Ausdruck, der mit dem Operator niedrigster Präzedenz gebildet wurde.

```
> Expr1 :: {CoreExpr}
>    : Expr2 '|' Expr1    { EAp (EAp (EVar "|") $1) $3 }
>    | Expr2              { $1 }
```

[1] Constant Applicative Form

[2] Formal gesehen handelt es sich hierbei um die Beseitigung von Mehrdeutigkeiten aus einer kontextfreien Grammatik, die eine eindeutige Sprache beschreibt.

Dies wird dann entsprechend der Präzedenztabelle aus [PJL91, Seite 7] fortgeführt, bis man bei atomaren Ausdrücken angelangt.

```
> Expr6 :: {CoreExpr}
>   : AExprList         { $1 }

> AExprList :: {CoreExpr}
>   : AExpr           { $1 }
>   | AExprList AExpr  { EAp $1 $2 }

> AExpr :: {CoreExpr}
>   : var                       { EVar $1 }
>   | num                       { ENum $1 }
>   | char                      { makeCoreExprOfChar $1 }
>   | str                       { makeCoreExprOfStr $1 }
>   | pack '{' num ',' num '}'  { EConstr $3 $5 }
>   | '(' Expr ')'              { $2 }
```

Zuletzt werden noch die Alternativen für ein `case` spezifiziert.

```
> Alts :: {[CoreAlt]}
>   : Alt          { [$1] }
>   | Alt ';' Alts  { $1 : $3 }
> Alt :: {CoreAlt}
>   : '<' num '>' '-' '>' Expr                { ($2, [], $6) }
>   | '<' num '>' VarNamesList '-' '>' Expr   { ($2, $4, $7) }
```

Es folgt nun ein Nachspann, in dem vom Parser benutzte Funktionen definiert werden können. Da wäre zum Beispiel die Funktion `luckyError`, die im Fehlerfall vom Parser aufgerufen wird.

```
> {

> luckyError :: [Token] -> coreProgram
> luckyError [] = error "unexspected EOF"
> luckyError ((lexem, line, col):_) =
>    error ("(" ++ show line ++ ", " ++ show col ++
>       ") syntax error on input " ++ show lexem)
```

Außerdem erlauben wir in den Kernsprachenprogrammen die Angabe von Zeichen und Zeichenketten als Literale. Diese müssen also erst auf algebraische Datentypen umgesetzt werden, was mittels `makeCoreExprOfChar` und `makeCoreExprOfString` geschieht.

```
> makeCoreExprOfChar :: String -> CoreExpr
> makeCoreExprOfChar ('\\':'^':c:[]) =
>   EConstr ((fromEnum c) - 64) 0
> makeCoreExprOfChar ('\\':c:[])
>   | c `elem` "\\\'nr" = EConstr (fromEnum c) 0

> makeCoreExprOfChar (c:[]) = EConstr (fromEnum c) 0
> makeCoreExprOfChar _      = error "illegal char literal"


> makeCoreExprOfStr :: String -> CoreExpr
> makeCoreExprOfStr [] = nil
>   where
>     nil = EConstr 1 0

> makeCoreExprOfStr (c:cs) =
>   EAp (EAp cons (makeCoreExprOfChar [c]))
>       (makeCoreExprOfStr cs)
>   where
>     cons = EConstr 2 2
```

Nun ist die Parsefunktion selbst nichts weiter als die Komposition aus `lexit` und `syntax`.

```
> type Parser a = [Token] -> [(a, [Token])]

> syntax :: [Token] -> CoreProgram

> parse :: String -> CoreProgram
> parse = syntax . lexit
```

Der Lexer wird hier aus Platzgründen nicht angegeben. Er ist keineswegs kompliziert, nur wegen zahlreicher Fallunterscheidungen zu umfangreich. Angemerkt sei, daß er nicht nur die Aufgabe erfüllt, das Quellprogramm in die Lexeme zu zerlegen, sondern dabei auch Zeilen und Spalten zählt und in den Token ablegt. Schließlich endet der Nachspann mit:

```
> }
```

4.2 Operationale Semantik

In diesem Abschnitt wird nun beschrieben, wie Ausdrücke aus Λ_C ausgewertet werden sollen, indem die Reduktionsrelation $\to_\delta$ definiert wird. Bei allen Betrachtungen fassen wir `bot` als den Namen eines Superkombinators auf, der in jedem Kernsprachenprogramm durch die Gleichung `bot` = `bot` definiert sei. Außerdem wird noch der Begriff eines Kontextes benötigt.

Definition 4.2.1. Einen *Term-* bzw. *Programmkontext*, kurz *Kontext*, $P[\cdot]$ definieren wir leicht abweichend[3] von Definition 2.2.2 als einen Term mit *einem einzigen* Loch. $P[t]$ ist dann der Ausdruck, der durch das Einsetzen von t an der Stelle des Lochs entsteht.

Notation. Wie bereits in Abschnitt 2.2 schreiben wir $P[\cdot] \in \Lambda_C$ dafür, daß $P[\cdot]$ ein Kontext über Λ_C ist, also $t \in \Lambda_C \implies P[t] \in \Lambda_C$.

4.2.1 Reduktion

Nun wird zuerst eine Relation $\to_B$ eingeführt, mit deren Hilfe später die Reduktionsrelation $\to_\delta$ angegeben werden kann.

Definition 4.2.2. Sei P ein Kernsprachenprogramm, welches den Superkombinator `f` durch `f` $x_1 \ \ldots \ x_n = t$ definiere. Die Reduktion einer Anwendung von `f` auf Terme e_i mit $1 \le i \le n$ ist erklärt durch die folgende Definitionsexpansion:

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

Neben dieser Superkombinatorreduktion gilt es, auch die Anwendung von `case` zu betrachten.

Definition 4.2.3. Die Anwendung von `case` auf ein Datenobjekt wird foldendermaßen reduziert:

$$\texttt{case Cons}\{j,m\}\ t_1 \ \ldots \ t_m \ \texttt{of}\ a_1;\ \ldots;\ a_n \ \to_B \ e_j[t_1/x_1, \ldots, t_m/x_m]$$

für eine Alternative der Form $a_j \equiv$ `<`j`>` $x_1 \ \ldots \ x_m$ `->` e_j.

Definition 4.2.4. Gilt $s \to_B t$, kann s also mit $\to_B$ auf t reduziert werden, so heißt s *Redex* und t *Redukt*. Diese Bezeichnung wird auch verwendet, wenn s und t selbst Unterterme eines Ausdruckes $P[s]$ resp. $P[t]$ sind (vgl. dazu auch die Definitionen 2.3.4 und 2.3.5).

[3] Deswegen wurde auch die andere Notation $P[\cdot]$ anstelle von $C[\ ]$ gewählt.

Jetzt können wir die Reduktionsrelation $\rightarrow_\delta$ bequem über Kontexte erklären.

Definition 4.2.5. Die Reduktionsrelation $\rightarrow_\delta \subseteq \Lambda_C \times \Lambda_C$ ist die kleinste Relation, so daß für beliebige Kontexte $P[\cdot]$ gilt:

$$s \rightarrow_B t \Longrightarrow P[s] \rightarrow_\delta P[t] \tag{4.1}$$

Für die reflexiv-transitive Hülle von $\rightarrow_\delta$ steht $\stackrel{*}{\rightarrow}_\delta$. Beide Relationen bezeichnen wir als *Reduktion*, $\rightarrow_\delta$ manchmal zur Unterscheidung auch als *Reduktionsschritt*.

Korollar 4.2.1. *Sei $P[\cdot]$ ein Programmkontext, so gilt*

$$P[\texttt{bot}] \rightarrow_\delta P[\texttt{bot}]$$

Beweis. Durch die Definition $\texttt{bot} = \texttt{bot}$ erhalten wir wegen $\texttt{bot} \rightarrow_B \texttt{bot}$ sofort die Behauptung. □

Weiterhin läßt sich die Auswertung in Normalordnung leicht formalisieren, wenn man wie in [Sch] *Reduktionskontexte* benutzt.

Definition 4.2.6. Ein *Reduktionskontext* ist ein Kontext, der durch folgende Grammatik definiert wird:

$$\begin{aligned} R &\rightarrow [\cdot] \\ &\mid R\ e \\ &\mid \texttt{case}\ R\ \texttt{of}\ a_1;\ \ldots;\ a_n \end{aligned}$$

Ist R ein Reduktionskontext, so bezeichnen wir $R[s] \rightarrow_\delta R[t]$ als *notwendigen Reduktionsschritt* und s als *notwendigen Redex*. Allgemein spricht man auch davon, daß sich e in $R[e]$ an einer *notwendigen Position* befindet.

Definition 4.2.7. Eine Reduktion ist in *normaler Ordnung*, wenn Gleichung (4.1) für einen bzw. eine Folge von Reduktionskontexten gilt. Wir schreiben dann $\rightarrow_\delta^{\text{no}}$ bzw. $\stackrel{*}{\rightarrow}_\delta^{\text{no}}$.

Bemerkung. Eine Reduktion in normaler Ordnung besteht also nur aus notwendigen Reduktionsschritten.

Lemma 4.2.2. *Die Relation $\rightarrow_\delta$ ist konfluent, d.h.*

$$q \stackrel{*}{\rightarrow}_\delta r \ \wedge\ q \stackrel{*}{\rightarrow}_\delta s \Longrightarrow (\exists t)\ r \stackrel{*}{\rightarrow}_\delta t \ \wedge\ s \stackrel{*}{\rightarrow}_\delta t$$

Beweis. Siehe [Sch], da $\rightarrow_\delta$ identisch zu dort definiert ist. □

4.2.2 Eine andere Form für `case`

In der Syntax der Kernsprache gibt es nur ein einziges `case`-Konstrukt, daß für alle algebraischen Typen A verwendet wird. Für die Gleichheitsanalyse müssen wir verlangen, daß ein `case` in einem Ausdruck immer *vollständig spezifiziert* ist, d.h. Alternativen für alle Konstruktoren des jeweiligen Typs angegeben sind. Somit kann jedes `case` auch als ein spezielles $\texttt{case}_A$ für einen algebraischen Datentypen A betrachtet werden, oder anders: Jedes `case` stellt eine bestimmte $\texttt{case}_A$-Konstante für einen algebraischen Datentypen A dar.

Theoretische Erörterungen werden manchmal unnötig dadurch verkompliziert, daß das bisherige `case`-Konstrukt neue Variablen einführt, die in den Ausdrücken für die Alternativen auftauchen dürfen. Um dies umgehen zu können, betrachten wir auch noch eine alternative Form mit einer $\texttt{case}_A$-Konstante für jeden algebraischen Typen. Deren Syntax ist durch die Regel

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

gegeben. Es sei hervorgehoben, daß diese Form *nicht* vom Parser erkannt wird, sondern lediglich theoretische Betrachtungen erleichtern soll.

Selbstverständlich sollen beide Sprachkonstrukte semantisch gleichwertig sein, d.h. angewendet auf die gleichen Argumente auch das gleiche Ergebnis liefern. Dazu ist es nötig, die Reduktion $\to_B$ auf $\texttt{case}_A$ entsprechend zu erklären.

Definition 4.2.8. Zur Relation $\to_B$ wird folgende Möglichkeit hinzugefügt:

$$\texttt{case}_A \; (\texttt{con}_{A,i} \; t_1 \; \ldots \; t_{ar(\texttt{con}_{A,i})}) \; e_1 \; \ldots \; e_{|A|} \to_B \; e_i \; t_1 \; \ldots \; t_{ar(\texttt{con}_{A,i})}$$

Nun ist es relativ offensichtlich, daß beide Konstrukte äquivalent sind.

Definition 4.2.9. Zu der $\texttt{case}_A$-Konstante jedes algebraischen Typen A bezeichne `scCaseA` den durch

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{scCaseA} \; t \; f_1 \; \ldots \; f_{|A|} = \texttt{case} \; t \; \texttt{of}} \\
\texttt{<1>} & x_{\texttt{con}_{A,1},1} \; \cdots \; x_{\texttt{con}_{A,1},ar(\texttt{con}_{A,1})} \; \texttt{->} \\
 & f_1 \; x_{\texttt{con}_{A,1},1} \; \cdots \; x_{\texttt{con}_{A,1},ar(\texttt{con}_{A,1})} \\
 & \vdots \\
\texttt{<}|A|\texttt{>} & x_{\texttt{con}_{A,|A|},1} \; \cdots \; x_{\texttt{con}_{A,|A|},ar(\texttt{con}_{A,1})} \; \texttt{->} \\
 & f_{|A|} \; x_{\texttt{con}_{A,|A|},1} \; \cdots \; x_{\texttt{con}_{A,|A|},ar(\texttt{con}_{A,1})}
\end{array}$$

definierten Superkombinator, wobei alle $x_{i,k}$ neue Variablen sind.

Satz 4.2.3. *In einem Ausdruck kann jedes Auftreten einer* $\texttt{case}_A$*-Konstanten durch den nach Definition 4.2.9 zugehörigen Superkombinator* `scCaseA` *ersetzt werden, ohne daß sich die Bedeutung des Ausdruckes verändert.*

Beweis. Man sieht schnell, daß ein Ausdruck

$$\texttt{scCaseA}\ (\texttt{con}_{A,i}\ t_1\ \dots\ t_{ar(\texttt{con}_{A,i})})\ e_1\ \dots\ e_{|A|}$$

genauso wie

$$\texttt{case}_A(\texttt{con}_{A,i}\ t_1\ \dots\ t_{ar(\texttt{con}_{A,i})})\ e_1\ \dots\ e_{|A|}$$

zu $e_i\ t_1\ \dots\ t_{ar(\texttt{con}_{A,i})}$ reduziert. Andere Möglichkeiten zur Reduktion gibt es in beiden Fällen nicht. □

Satz 4.2.4. *Jeder Ausdruck* `case` t `of` $a_1; \dots; a_n$*, in dem die Alternativen die Form* $a_i \equiv$ `<i>` $x_{i,1}\ \dots\ x_{i,i_{ar(\texttt{con}_{A,i})}}$ `->` rhs_i *haben, kann auf folgende Weise auch mit* $\texttt{case}_A$ *dargestellt werden:*

$$\texttt{case}_A\ t\ \texttt{f}_1\ \dots\ \texttt{f}_\texttt{n}$$

Die $\texttt{f}_\texttt{i}$ *sind dabei neue Superkombinatoren, die durch die Gleichungen*

$$\texttt{f}_\texttt{i}\ x_{i,1}\ \dots\ x_{i,i_{ar(\texttt{con}_{A,i})}} = rhs_i$$

definiert werden.

Beweis. Auch hier sieht man schnell, daß die betreffenden Ausdrücke identisch reduzieren. □

Wir können also `case` ... `of` und $\texttt{case}_A$ nach Belieben benutzen.

4.2.3 Weitere Begriffe

Weitere Begriffe, die häufig gebraucht werden und daher wichtig sind, werden im folgenden erklärt.

Definition 4.2.10. Ein Ausdruck $t \in \Lambda_C$ ist in *schwacher Kopfnormalform*[4], kurz WHNF, falls er in einer der folgenden Formen ist:

- $t \equiv \texttt{con}\ e_1\ \dots\ e_n$, wenn `con` ein Konstruktor und $n \leq ar(\texttt{con})$.
- $t \equiv \texttt{f}\ e_1\ \dots\ e_n$, wenn `f` ein Superkombinator mit $n < ar(\texttt{f})$ oder $\texttt{case}_A$ und $n < |A| + 1$.

Eine WHNF der ersten Form, stellt eine *Konstruktor*-Anwendung dar. Daher nennen wir sie CWHNF[5]. Gilt $n = ar(\texttt{con})$, so ist die CWHNF *gesättigt* und heißt SCWHNF[6]. Ist sie *ungesättigt*, also $n < ar(\texttt{con})$, so wird sie auch — genauso wie eine WHNF der zweiten Form — als FWHNF[7] bezeichnet, da es sich um eine *Funktion* handelt. Ausdrücke ohne WHNF sieht man semantisch als bedeutungslos an.

[4] engl. *weak head normal form*
[5] engl. *constructor WHNF*
[6] engl. *saturated CWHNF*
[7] engl. *function WHNF*

4.3 Abstraktion von Λ_C

Mit der Gleichheitsanalyse sollen ja nicht nur Gleichheiten speziell zwischen zwei konkreten Termen gezeigt, sondern auch allgemeingültige Aussagen bewiesen werden. Dafür brauchen wir eine Möglichkeit, Mengen von Termen darzustellen. Diese schaffen wir, indem wir als weitere mögliche Ausdrücke freie[8] Variablen einführen und so die Kernsprache Λ_C aus dem vorigen Abschnitt abstrahieren. Die entsprechende Ergänzung zu den Regeln der Grammatik[9] ist in Abbildung 4.2 zu sehen. Die so definierte Abstraktion von

Atomarer Ausdruck	*aexpr*	$\rightarrow$	...	
		$\mid$	*absvar*	Abstraktion
Abstraktion	*absvar*	$\rightarrow$	*var*	

Abbildung 4.2: Erweiterung der Syntax zur Abstraktion von Λ_C

Λ_C bezeichnen wir als $\Lambda_C^\#$, die Menge der freien Variablen eines Ausdrucks $t \in \Lambda_C^\#$ analog zu Λ mit $\mathcal{FV}(t)$. Auch die Menge der geschlossen Terme könnte wieder durch

$$\Lambda_C^{\#^0} = \{t \in \Lambda_C^\# \mid \mathcal{FV}(t) = \emptyset\}$$

definiert werden; aber diese ist, wie man schnell sieht, identisch zu Λ_C.

Definition 4.3.1. Einen Term $t \in \Lambda_C$, der *keine Variablen* enthält, bezeichnen wir als *Grundterm.* Ein Grundterm, in dem kein Superkombinator benutzt wird, besteht nur aus Konstruktoren und heißt deswegen *Konstruktorgrundterm.*

Bemerkung. Da wir `case` ... `of` in $\texttt{case}_A$ übersetzen können, gibt es für jeden geschlossenen Term t, d.h. $t \in \Lambda_C$, einen Grundterm, der identisch reduziert. Daher sind die Begriffe Grundterm und geschlossener Term im Bezug auf $\Lambda_C^\#$ gleichwertig.

Es können also alle $t \in \Lambda_C$ als Grundterme angesehen werden. Jene Terme, die gerade durch die Abstraktion hinzugekommen sind, werden durch die Menge $\Lambda_C^\# \setminus \Lambda_C$ beschrieben. Daher heißt $t' \in \Lambda_C^\# \setminus \Lambda_C$ auch *abstrakter Ausdruck.* Wir möchten nun auch für solche abstrakten Ausdrücke eine Reduktion definieren, indem wir die Relation $\rightarrow_\delta$ auf $\Lambda_C^\# \times \Lambda_C^\#$ erweitern. Dies geschieht dadurch, daß in den Definitionen 4.2.4 und 4.2.5 einfach $s, t \in \Lambda_C^\#$ angenommen wird. Kontexte werden dabei über $\Lambda_C^\#$ aufgefaßt, wofür wieder die schon eingeführte Notation $P[\cdot] \in \Lambda_C^\#$ benutzt wird.

[8] Diese stehen im Gegensatz zu den in den Alternativen eines `case` gebundenen.

[9] Die Grammatik dient hier lediglich zur Illustration, da die Unterscheidung zwischen freien und gebundenen Variablen nicht mehr kontextfrei ist.

Definition 4.3.2. Für $s, t \in \Lambda_C^{\#}$ gilt $s \rightarrow_\delta t$ genau dann, wenn (4.1) mit einem beliebigen Kontext $P[\cdot] \in \Lambda_C^{\#}$ erfüllt ist. Wir nennen dies eine *abstrakte Reduktion* und der Zusatz ^{no} wird entsprechend für die Auswerung in normaler Ordnung verwendet.

Das bedeutet, die Reduktion ist auf Grundtermen wie bisher erklärt, und es gibt auch keine gesonderte Behandlung der Variablen.

4.4 Gleichheit

In diesem Abschnitt soll erklärt werden, wann wir zwei Kernsprachenprogramme als gleich betrachten möchten. Es wird dann anschließend die Gleichheit auf den Grundtermen der Kernsprache definiert.

Grundsätzlich soll es sich, genauso wie beim Ansatz von Abramsky, um eine *Verhaltensgleichheit* handeln. Das heißt, zwei Programme s und t sollen genau dann als gleich gelten, wenn sie, eingesetzt in irgendeinen Programmkontext $P[\cdot]$, das gleiche Verhalten zeigen. Es liegt auf der Hand, daß bei einer Sprache mit algebraischen Datentypen die Beobachtung von Terminierung bzw. Nichtterminierung nicht das Entscheidende sein kann. Vielmehr interessiert die Gestalt eines Datenobjektes, insbesondere also der Konstruktor in einer SCWHNF.

In einem ersten Ansatz wird man daher intuitiv zwei Ausdrücke s und t als gleich bezeichnen, wenn für alle möglichen Programmkontexte $P[\cdot]$ der Ausdruck $P[s]$ zu einer SCWHNF $\texttt{con}\ s_1\ \ldots\ s_{ar(\texttt{con})}$ reduziert, genau dann wenn $P[t]$ zu einer SCWHNF $\texttt{con}\ t_1\ \ldots\ t_{ar(\texttt{con})}$ reduziert.

4.4.1 Wohlgetyptheit von Ausdrücken

Es bleibt allerdings zu untersuchen, inwieweit dabei dem Aspekt wohlgetypter Terme Rechnung getragen wird. Da wir über algebraische Datentypen mit verschiedenen Konstruktoren verfügen, gibt es auch Ausdrücke, die nicht sinnvoll sind. Welche dies sind, soll nun im einzelnen behandelt werden.

Übersättigte Konstruktoren

In Abschnitt 4.1.1 erfolgte die Begriffsbildung in der Art, daß jeder Konstruktor eine feste Stelligkeit besitzt. Das bedeutet, daß er höchstens auf eine ganz bestimmte Anzahl von Argumenten angewendet werden kann. Diese Forderung ist deswegen sinnvoll, weil sonst die einzige Alternative darin bestünde, die Anwendung aller Konstruktoren auf unendlich viele Argumente zuzulassen.

Terme, in denen die maximale Anzahl von Argumenten eines Konstruktors überschritten wird, nennt man *übersättigte* Konstruktoranwendungen oder einfach übersättigte Konstruktoren. Diese haben allgemein die Form

$\mathtt{con}\ e_1\ \ldots\ e_n$ mit $n > ar(\mathtt{con})$ und müssen konsequenterweise als *nicht wohlgetypt* eingeordnet werden, da sie nicht für ein zulässiges Datenobjekt stehen.

Unpassendes `case`

Aus Definition 4.2.3 und der Forderung nach einer vollständigen Spezifikation jedes `case...of` erkennt man sofort, daß dessen Reduktion nur im Falle einer Anwendung auf eine SCWHNF passenden Typs erklärt ist.

Auch auf die Reduktionsregel für $\mathtt{case}_A$ aus Abschnitt 4.2.2 trifft dies zu, d.h. eine Reduktion ist nur auf Ausdrücken der Form

$$\mathtt{case}_A\ (\mathtt{con}_{A,i}\ t_1\ \ldots\ t_{ar(\mathtt{con}_{A,i})})\ e_1\ \ldots\ e_{|A|}$$

möglich. Welches Ergebnis sollte denn auch die Anwendung eines $\mathtt{case}_A$ auf einen Konstruktor eines Typs $B \neq A$ geben? Wir möchten vermeiden, für solche Terme eine Reduktion angeben zu müssen, da sie wiederum nicht für zulässige Datenobjekte stehen. Präzisiert man obige Ausführungen, so sind alle Terme der Form

$$\mathtt{case}_A\ (\mathtt{con}_{B,i}\ t_1\ \ldots\ t_{ar(\mathtt{con}_{A,i})})\ e_1\ \ldots\ e_{|A|}$$

mit $A \neq B$ nicht wohlgetypt, genauso wie solche der Form

$$\mathtt{case}_A\ t\ e_1\ \ldots\ e_{|A|}$$

falls t eine FWHNF hat.

Die Menge wohlgetypter Terme

Bei den Termen, die durch die soeben erfolgten Betrachtungen als nicht wohlgetypt gelten, ist das auch ohne weiteres sofort ersichtlich. Daher erhalten diese eine eigene Bezeichnung.

Definition 4.4.1. Wir definieren die Menge $DNWT$[10] der *direkt nicht wohlgetypten Terme* durch

$$\begin{aligned} DNWT = \{\ t \in \Lambda_C \mid\ & (t \equiv \mathtt{con}\ e_1\ \ldots\ e_n\ \wedge\ n > ar(\mathtt{con})) \\ & \vee (t \equiv \mathtt{case}_A\ (\mathtt{con}_{B,i}\ t_1\ \ldots\ t_{ar(\mathtt{con}_{A,i})})\ e_1\ \ldots\ e_{|A|})\ \wedge\ A \neq B \\ & \vee (t \equiv \mathtt{case}_A\ t\ e_1\ \ldots\ e_{|A|}\ \wedge t \text{ ist eine FWHNF})\ \} \end{aligned}$$

Insgesamt möchten wir, daß die Wohlgetyptheit mit der Reduktion konform geht. Das bedeutet, nicht wohlgetypt sollen all die Terme sein, in denen die Auswertung in normaler Ordnung auf einen direkt nicht wohlgetypten Unterausdruck stößt.

[10] engl. *directly non-well-typed* (with d, n, w, t underlined)

Definition 4.4.2. Wir definieren die Menge NWT der *nicht wohlgetypten Ausdrücke* aus Λ_C folgendermaßen:

$$NWT = \{s \mid s \xrightarrow{*}_{\delta}^{\text{no}} R[t], \text{ R ist Reduktionskontext und } t \in DNWT\ \}$$

Nun ist es sehr einfach anzugeben, welche Ausdrücke wohlgetypt sind.

Definition 4.4.3. Wir defineren die Menge WT der *wohlgetypten Ausdrücke* aus Λ_C entsprechend

$$WT = \Lambda_C \setminus NWT$$

4.4.2 Vorüberlegungen zur kontextuellen Gleichheit

Um nun eine geeignete Definition für die kontextuelle Gleichheit zu finden, bei der die Wohlgetyptheit angemessen berücksichtigt wird, waren daher sehr sorgfältige Überlegungen notwendig. Diese sollen im folgenden ausführlich geschildert und anschaulich begründet werden.

Denn an sich spielt es für die Verhaltensgleichheit von Ausdrücken eine Rolle, ob nicht-wohlgetypt mit bedeutungslos identifiziert wird. Denn bei der Ausführung in einem Interpreter wird ein nicht wohlgetyptes Programm in der Regel zu einem Laufzeitfehler führen, wohingegen ein bedeutungsloses Programm nicht terminiert.

Anderseits haben wir ja gesehen, daß ein nicht wohlgetypter Ausdruck niemals für ein gültiges Datenobjekt stehen kann. Da wir uns aber gerade für die Struktur der Datenobjekte interessieren, können wir die nicht wohlgetypten Terme berechtigterweise als bedeutungslos einordnen.

Zumindest sollten nicht-wohlgetypte Terme also untereinander austauschbar sein. Damit ergeben sich für den Fall, daß in einem Programm $P[s]$ der Ausdruck s durch t ersetzt werden soll, schon zwei Forderungen:

1. $P[s]$ ist wohlgetypt *genau dann, wenn* $P[t]$ wohlgetypt ist.

2. $P[s]$ hat eine WHNF *genau dann, wenn* $P[t]$ eine WHNF hat.

Auf der anderen Seite möchten wir natürlich auch das gleiche Datenobjekt erhalten; wir hatten dies ja schon einmal in der Einleitung zum Abschnitt 4.4 formuliert:

3. $P[s]$ hat eine SCWHNF $\texttt{con}\ s_1\ \dots\ s_{ar(\texttt{con})}$ *genau dann, wenn* $P[t]$ eine SCWHNF $\texttt{con}\ t_1\ \dots\ t_{ar(\texttt{con})}$ hat.

Darauf, daß wir diese Bedingungen für beliebige Programmkontexte $P[\cdot]$ fordern, gründen sich die ersten Folgerungen.

Betrachtung auf oberster Ebene

Wir müssen für die Terme $s_1, \ldots, s_n$ aus der SCWHNF nicht explizit verlangen, daß sie zu $t_1, \ldots, t_n$ kontextuell gleich sind, da dies implizit in der 3. Bedingung enthalten ist. Denn wir können ohne weiteres einen Kontext $P'[\cdot]$ konstruieren, mit dem über ein passendes[11] `case` die i-te Komponente der SCWHNF, also s_i resp. t_i, isoliert wird. Es genügt also, wenn zum Vergleich der Konstruktor auf der obersten Ebene einer SCWHNF — wir nennen diesen *Toplevelkonstruktor* — herangezogen wird.

WHNF und SCWHNF

Bei der Betrachtung *aller* möglichen Kontexte besteht der Unterschied zwischen WHNF und SCWHNF im wesentlichen darin, ob wir folgende Terme als bedeutungslos ansehen möchten:

- $(\boldsymbol{Y}\ \boldsymbol{K})$ mitsamt der nach Eigenschaft 2 dazu äquivalenten Terme [12]
- Superkombinatoren `f`, die durch `f` $x_1 \ \ldots \ x_n = t$ definiert sind, für die es aber keine $e_1, \ldots, e_n \in \Lambda_C$ gibt, so daß $t[e_1/x_1, \ldots, e_n/x_n]$ eine WHNF hat, die nicht $(\boldsymbol{Y}\ \boldsymbol{K})$ ist[13].

Daß die 2. und 3. Bedingung ohne die Berücksichtigung solcher Terme sonst nämlich gleichwertig wären, zeigt folgender

Versuch: Seien s und t zwei Kernsprachenausdrücke, und für jeden beliebigen Kontext $P[\cdot]$ gelte, daß $P[s]$ eine SCWHNF $\texttt{con}\ s_1 \ \ldots \ s_{ar(\texttt{con})}$ hat gdw. $P[t]$ eine SCWHNF $\texttt{con}\ t_1 \ \ldots \ t_{ar(\texttt{con})}$ hat. Daraus kann man zu folgern *versuchen*, daß für jeden beliebigen Kontext $P[\cdot]$ gilt: $P[s]$ hat eine WHNF gdw. $P[t]$ eine WHNF hat.

Wir nehmen dafür an, daß es einen Kontext $P[\cdot]$ gibt, so daß $P[s]$ eine WHNF hat und $P[t]$ nicht, oder umgekehrt. O.B.d.A. betrachten wir nur die erste Möglichkeit, da sich die zweite symmetrisch verhält. Dabei können wir folgende Fälle unterscheiden:

1. $P[s]$ hat eine SCWHNF. Dies kann nach Voraussetzung nicht sein, da dann auch $P[t]$ eine SCWHNF mit dem gleichen Toplevelkonstruktor hätte.

[11] D.h. es handelt sich um $\texttt{case}_A$ wenn `con` ein Konstruktor des algebraischen Typs A ist, also $\texttt{con}_A$.

[12] Dies ist gerade die der kontextuellen Ordnung $\leq_c$ aus 2.4.1 zugrunde liegende Gleichheit. Wir betrachten $(\boldsymbol{Y}\ \boldsymbol{K})$ im folgenden stets modulo dieser Äquivalenz und behalten daher auch die bisherige Typographie für $(\boldsymbol{Y}\ \boldsymbol{K})$ bei. Da wir eine Menge von Ausdrücken beschreiben, benutzen wir also nicht (`Y K`) für zwei in der Kernsprache definierte Superkombinatoren `Y` $f = f$ (`Y` f) und `K` $x\ y = x$.

[13] bzw. äquivalent hierzu, s.o.

2. $P[s]$ hat eine FWHNF.

   (a) $P[s]$ hat eine (ungesättigte) CWHNF. Nun kann man aus $P[\cdot]$ einen Kontext $P'[\cdot]$ konstruieren, indem man die fehlenden Argumente ergänzt, so daß $P'[s]$ eine SCWHNF hat. Da das Ergänzen der Argumente eine Applikation darstellt, hat $P'[t]$ damit (immer noch) keine WHNF. Dies bedeutet also einen Widerspruch zur Voraussetzung.

   (b) $P[s]$ hat als FWHNF einen ungesättigten Superkombinatorausdruck $\mathtt{sc}\ s_1\ \ldots\ s_n$ mit $n < ar(\mathtt{sc})$. Aus $P[\cdot]$ könnten wir wiederum einen Kontext $P'[\cdot]$ konstruieren, der die fehlenden Argumente ergänzt. Danach sind mit der Definitionsexpansion weitere Reduktionsschritte möglich.

   Wenn wir nun die o.g. Terme ausschließen dürften, die entweder unendlich oft „ein Argument konsumieren“ können oder die besagten n-stelligen Funktionen darstellen, so könnten wir an dieser Stelle induktiv vorgehen. Das heißt, wir hätten nach endlich vielen Schritten einen Kontext $P^{(n)}[\cdot]$ konstruiert, so daß $P^{(n)}[s]$ eine SCWHNF und $P^{(n)}[t]$ keine WHNF hat.

   Ohne Einschränkungen ist dies aber nicht möglich, so daß unser Versuch an dieser Stelle scheitert.

Mit n-stelligen Funktionen, die bei keiner Anwendung zu irgendeiner WHNF reduzieren, könnte noch recht einfach in der Weise umgegangen werden, wie es die Definition 2.4.6 beschreibt.

Wir sehen also, daß die wirklichen Problemfälle in unserem Versuch gerade die Terme sind, die wie $(\boldsymbol{Y}\ \boldsymbol{K})$ eben diesen unendlichen Prozess, vgl. [Abr90, Seite 72], darstellen, immer wieder auf ein Argument angewendet werden zu können und dabei zu einer WHNF zu reduzieren. Ausführlich dargestellt wird im folgenden die sich hieraus ergebende

Beurteilung der Frage, ob $(\boldsymbol{Y}\ \boldsymbol{K})$ als bedeutungslos betrachtet werden soll. In der Ordnung von Abramsky ist $(\boldsymbol{Y}\ \boldsymbol{K})$ immerhin das größte Element, weil es in allen Stufen der Beobachtung eine WHNF hat. Da es sich bei dieser WHNF aber in jedem Fall um einen ungesättigten Superkombinatorausdruck handelt, ist sie für unsere Zwecke nicht sehr bedeutungsvoll. Denn wir interessieren uns ja gerade für die Gestalt von Datenobjekten; diese werden stets durch eine CWHNF charakterisiert. Somit bleibt festzustellen, daß $(\boldsymbol{Y}\ \boldsymbol{K})$ zwar immer und immer wieder auf ein Argument angewendet werden kann, die hierdurch entstandenen Terme aber niemals zu einem Datenobjekt reduzieren.

Man kann damit eine zum „Genericity lemma“ aus [Bar84, Seite 374] ähnliche Aussage formulieren: Sei $P[\cdot]$ ein beliebiger Programmkontext und $s \equiv P[(\boldsymbol{Y}\ \boldsymbol{K})]$ ein Ausdruck mit einer CWHNF, so hat auch $s' \equiv P[t]$

für alle möglichen Terme t eine CWHNF mit demselben Toplevelkonstruktor. Da diese und andere Aussagen in Abschnitt 4.4.3 noch einmal genauer untersucht werden sollen, wird ihre Begründung hier nur kurz angedeutet: Angenommen $(\boldsymbol{Y}\ \boldsymbol{K})$ ist in $P[(\boldsymbol{Y}\ \boldsymbol{K})]$ ein notwendiger Redex, dann hat s nach dem oben gesagten keine CWHNF. Ist $(\boldsymbol{Y}\ \boldsymbol{K})$ in $P[(\boldsymbol{Y}\ \boldsymbol{K})]$ kein notwendiger Redex, so wird die Reduktion zu CWHNF hiervon auch nicht beeinflußt[14], d.h. s' konvergiert zur selben CWHNF wie s.

Nun erscheint es berechtigt, $(\boldsymbol{Y}\ \boldsymbol{K})$ in einem beliebigen Programm P durch einen Term ohne WHNF, resp. `bot`, zu ersetzen. Das heißt, wir setzen $(\boldsymbol{Y}\ \boldsymbol{K})$ mit bedeutunglosen Termen gleich.

4.4.3 Konvergenz und ihre Eigenschaften

In diesem Abschnitt soll ein Begriff von *Konvergenz* definiert und seine wichtigsten Eigenschaften im Hinblick auf die kontextuelle Gleichheit untersucht werden.

Definition 4.4.4. Die Relation „s reduziert bzw. *konvergiert* zur CWHNF t" — in Zeichen $s \Downarrow t$ — wird für geschlossene Kernsprachenausdrücke $s, t \in \Lambda_C^0$ definiert durch

$$s \Downarrow t \text{ gdw. } s \text{ hat die CWHNF } t \equiv \texttt{con}\ t_1\ \ldots\ t_n$$

Wir schreiben kurz $s \Downarrow$, falls $(\exists t)\ s \Downarrow t$ und $s \Uparrow$ für das Gegenteil, d.h. wenn „s divergiert".

Welche Ausdrücke konvergieren nicht zu einer CWHNF? Um diese Frage drehen sich die folgenden Betrachtungen.

Korollar 4.4.1. *Hat $s \in \Lambda_C$ keine WHNF, so gilt $s \Uparrow$.*

Beweis. s hat keine WHNF und so erst recht keine CWHNF. ☐

Welche Rolle die nicht wohlgetypten Termen spielen, wird nun geklärt.

Korollar 4.4.2. *Sei $s \in NWT$, so gilt $s \Uparrow$.*

Beweis. Wegen $s \in NWT$ gibt es einen Reduktionskontext $R[\cdot]$, so daß $s \xrightarrow{*}{}_{\delta}^{no} R[t]$ und $t \in DNWT$. Nach der Definition eines Reduktionskontextes und der Reduktionsrelation $\xrightarrow{*}{}_{\delta}^{no}$ kann $R[t]$ weder eine CWHNF sein, noch sind weitere Reduktionsschritte in normaler Ordnung möglich. ☐

Bemerkung. Auch der Umkehrschluß $t \Downarrow \implies t \in WT$ ist nützlich.

[14] Auch wenn $(\boldsymbol{Y}\ \boldsymbol{K})$ selbstverständlich *in* dieser CWHNF vorkommen kann. Dann nämlich unterscheiden sich $P[(\boldsymbol{Y}\ \boldsymbol{K})]$ und $P[t]$ eben erst „unterhalb" des Konstruktors.

Im folgenden möchten wir die Notation $s \Downarrow_S t$ für eine Relation verwenden, bei der in Definition 4.4.4 die SCWHNF an Stelle der CWHNF steht[15]. $\Uparrow_S$ wird dann entsprechend benutzt. Nun gilt es, eine Reihe von Äquivalenzen zu zeigen.

Lemma 4.4.3. *Folgende Aussagen sind gleichwertig:*

1. $(\forall P[\cdot])\ P[s] \Downarrow \Longrightarrow\ P[t] \Downarrow$
2. $(\forall P[\cdot])\ P[s] \Downarrow \mathtt{con}\ s_1\ \dots\ s_n\ \Longrightarrow\ P[t] \Downarrow \mathtt{con}\ t_1\ \dots\ t_n$
3. $(\forall P[\cdot])\ P[s] \Downarrow_S \Longrightarrow\ P[t] \Downarrow_S$
4. $(\forall P[\cdot])\ P[s] \Downarrow_S \mathtt{con}\ s_1\ \dots\ s_{ar(\mathtt{con})}\ \Longrightarrow\ P[t] \Downarrow_S \mathtt{con}\ t_1\ \dots\ t_{ar(\mathtt{con})}$

Beweis. (1) $\Longrightarrow$ (3) Angenommen, es gelte (1) und $P[\cdot]$ sei ein beliebiger Programmkontext. Hat $P[s]$ keine CWHNF, so natürlich auch keine SCWHNF, und die Behauptung gilt. Haben $P[s]$ und $P[t]$ beide eine SCWHNF, so ist wiederum nichts zu zeigen. Nehmen wir also an, $P[s]$ habe eine SCWHNF und $P[t]$ eine ungesättigte CWHNF. Dann kann man einen Programmkontext der Form

$$P'[\cdot]\ \equiv\ \mathtt{case}_A\ P[\cdot]\ e_1\ \dots\ e_{|A|}$$

konstruieren, so daß $P'[s]$ in jeden Fall eine CWHNF hat. Dazu müssen nur die e_i geeignet gewählt werden. $P'[t]$ hingegen ist direkt nicht wohlgetypt, weil das $\mathtt{case}_A$ auf eine FWHNF angewendet wird. Dies bedeutet $P'[s] \Downarrow$ aber $P'[t] \Uparrow$, was einen Widerspruch zur Voraussetzung darstellt.

(3) $\Longrightarrow$ (4) Gelte (3) und sei $P[\cdot]$ ein beliebiger Programmkontext. Hat $P[s]$ keine SCWHNF so ist die Behauptung erfüllt. Haben $P[s]$ und $P[t]$ beide eine SCWHNF, so müssen wir mehrere Fälle unterscheiden. Sind die Toplevelkonstruktoren identisch, so gilt die Behauptung. Für unterschiedliche Konstruktoren nehmen wir $s \Downarrow_S \mathtt{con}_{A,i}\ s_1\ \dots\ s_{ar(\mathtt{con}_{A,i})}$ und $t \Downarrow_S \mathtt{con}_{B,j}\ t_1 \dots t_{ar(\mathtt{con}_{B,j})}$ an. Dann konstruieren wir einen Kontext

$$P'[\cdot]\ \equiv\ \mathtt{case}_A\ P[\cdot]\ e_1\ \dots\ e_{|A|}$$

Gehören die Konstruktoren $\mathtt{con}_{A,i}$ und $\mathtt{con}_{B,j}$ zu *unterschiedlichen* Typen, d.h. $A \neq B$, so ergibt sich sofort $P'[s] \Downarrow_S$ und $P'[t] \Uparrow_S$ im Widerspruch zur Voraussetzung. Mögen also beide Konstruktoren von einem Typ sein, also $A = B$, dann können sie sich nur durch $i \neq j$ unterscheiden. Wählt man e_i und e_j so, daß einerseits $e_i\ s_1\ \dots\ s_{ar(\mathtt{con}_{A,i})} \Downarrow_S$ aber $e_j\ t_1 \dots t_{ar(\mathtt{con}_{B,j})} \Uparrow_S$, so erhält man abermals $P'[s] \Downarrow_S$ und $P'[t] \Uparrow_S$ im Widerspruch zur Voraussetzung.

[15] Darunter verstehen wir natürlich, daß $n = ar(\mathtt{con})$ gilt. Man spricht dann von „Konvergenz zu SCWHNF“.

(4) $\Longrightarrow$ (2) soll über $\neg(2) \Longrightarrow \neg(4)$ gezeigt werden. Wir nehmen also an, daß es einen Programmkontext $P[\cdot]$ gibt, so daß zwar

$$P[s] \Downarrow \mathtt{con}\ s_1\ \dots\ s_n$$

gilt, nicht aber

$$P[t] \Downarrow \mathtt{con}\ t_1\ \dots\ t_n$$

Falls $n = ar(\mathtt{con})$, d.h. $\mathtt{con}\ s_1\ \dots\ s_n$ eine SCWHNF ist, so ist nichts zu beweisen. Sei also $n < ar(\mathtt{con})$, dann wählen wir

$$P'[\cdot] \equiv P[\cdot]\ a_1\ \dots\ a_{ar(\mathtt{con})-n}$$

mit $a_1\ \dots\ a_{ar(\mathtt{con})-n} \in \Lambda_C^0$. Somit ist die Behauptung bewiesen, da

$$P'[s] \Downarrow_S \mathtt{con}\ s_1\ \dots\ s_n\ a_1\ \dots\ a_{ar(\mathtt{con})-n}$$

gilt und

$$P'[t] \Downarrow_S \mathtt{con}\ t_1\ \dots\ t_n\ a_1\ \dots\ a_{ar(\mathtt{con})-n}$$

hingegen nicht. Denn die Voraussetzung $\neg(P[t] \Downarrow \mathtt{con}\ t_1\ \dots\ t_n)$ bedeutet ja, daß

- $P[t]$ nicht wohlgetypt ist,
- $P[t]$ keine CWHNF hat oder
- $P[t] \Downarrow \mathtt{con}'\ t_1\ \dots\ t_m$, wobei $\mathtt{con} \neq \mathtt{con}'$ oder $n \neq m$.

und diese Eigenschaften lassen sich direkt auf $P'[t]$ übertragen.

(2) $\Longrightarrow$ (1) ist trivial, da (1) eine Abschwächung von (2) bedeutet. Wenn für alle Kontexte $P[\cdot]$ gilt: Falls $P[s]$ zu einer CWHNF $\mathtt{con}\ s_1\ \dots\ s_n$ konvergiert, so reduziert $P[t]$ zur CWHNF $\mathtt{con}\ t_1\ \dots\ t_n$. Dann gilt sicherlich auch für alle Kontexte, daß, falls $P[s]$ überhaupt zu einer CWHNF reduziert, auch $P[t]$ zu irgend einer CWHNF konvergiert.

□

Lemma 4.4.4. *Sei $s \in \Lambda_C$ und $P[\cdot]$ ein Programmkontext über Λ_C, der kein Reduktionskontext ist, so gilt*

$$P[s] \Downarrow \quad \Longleftrightarrow \quad (\forall t \in \Lambda_C)\ P[t] \Downarrow$$

Beweis. Die Implikation $P[s] \Uparrow \Longrightarrow (\forall t \in \Lambda_C)\ P[t] \Uparrow$ ist trivial. Also nehmen wir $P[s] \Downarrow$ an, d.h. $P[s] \Downarrow \mathtt{con}\ s_1\ \dots\ s_n$ mit $n \leq ar(\mathtt{con})$. Da s in $P[s]$ nicht an einer notwendigen Position steht, kommt das Ergebnis $\mathtt{con}\ s_1\ \dots\ s_n$ ohne Reduktion von s zustande. Daher kann es sein, daß s gar nicht darin vorkommt, oder höchstens als ein Unterterm in einem der s_i. Das gleiche gilt nun auch für das t an der Stelle des Lochs. □

Die etwas umständliche Ausdrucksweise hatte ihren Grund: Denn man kann nicht ohne weiteres aus der Tatsache, daß s in $P[s]$ keine notwendige Position bezeichnet, folgern, daß auch t in $P[t]$ nicht an einer notwendigen Position steht. Kommt t nämlich schon in $P[\cdot]$ vor, so kann es sich in $P[s]$ und damit auch in $P[t]$ durchaus an einer notwendigen Position befinden. Dieser Fall stört aber nicht, da sich $P[s]$ und $P[t]$ dann in diesem Bezug auch nicht durch die Konvergenz zu einer CWHNF unterscheiden können.

Nun können wir in Analogie zur Behauptung 14.3.24 aus [Bar84] das folgende Lemma formulieren.

Lemma 4.4.5. *Sei $s \in \Lambda_C$ ein Term mit einer der folgenden Eigenschaften:*

1. *$s \in NWT$.*
2. *s hat keine WHNF.*
3. *$s \equiv (\boldsymbol{Y}\ \boldsymbol{K})$.*

Für einen beliebigen Programmkontext $P[\cdot] \in \Lambda_C$ gilt dann

$$P[s] \Downarrow \quad \Longrightarrow \quad (\forall t \in \Lambda_C)\ P[t] \Downarrow$$

Beweis. Zu (1) und (2) genügt folgende Beobachtung: Ist s in $P[s]$ an einer notwendigen Position, so gilt $P[s] \Uparrow$, da entweder $P[s] \in NWT$ oder $P[s]$ keine WHNF hat. Falls s in $P[s]$ nicht an einer notwendigen Position steht, folgt die Behauptung nach Lemma 4.4.4.

Für (3) erfolgt der Beweis induktiv über die Größe und Struktur des Kontexts $P[\cdot]$. Falls $P[\cdot] \equiv\ \cdot$, so $P[s] \Uparrow$, da $(\boldsymbol{Y}\ \boldsymbol{K})$ zwar eine WHNF hat, aber keine CWHNF. Bleiben die Applikationen $P[\cdot] \equiv P'[\cdot]\ a$ bzw. $P[\cdot] \equiv a\ P'[\cdot]$ mit $a \in \Lambda_C^0$. Die Anwendung der Induktionshypothese auf $P'[\cdot]$ erbringt in diesen Fällen den Beweis der Behauptung genauso, wie für Programmkontexte der Form

$$P[\cdot]\ \equiv\ \texttt{case}_A\ P'[\cdot]$$

bzw.

$$P[\cdot]\ \equiv\ \texttt{case}_A\ u\ e_1\ \ldots\ e_{i-1}\ P'[\cdot]\ e_{i+1}\ \ldots\ e_{|A|}$$

□

4.4.4 Kontextuelle Ordnungsrelation

Wir werden nun eine Ordnungsrelation angeben, die wie im Abschnitt 4.4.2 diskutiert, Terme über ihr Verhalten bei der Einsetzung in Kontexte klassifiziert. Das Lemma 4.4.3 aus Abschnitt 4.4.3 legt nun die folgende Definition nahe.

Definition 4.4.5. Für zwei Grundterme s und t gelte die Relation „s ist kontextuell kleiner als t“ — in Zeichen $s \leq_{con} t$ — genau dann, wenn

$$(\forall P[\cdot] \in \Lambda_C)\ P[s] \Downarrow \quad \Longrightarrow\ P[t] \Downarrow$$

Man könnte diese Ordnungsrelation sprachlich dadurch in Worte fassen, daß ein Term t größer ist als ein Term s, wenn t in einen Kontext eingesetzt mindestens genauso oft wohlgetypt ist und zu einer CWHNF führt wie s.

Als erstes zeigen wir, daß die Definition sinnvoll gewählt ist in dem Sinne, daß durch $\leq_{con}$ eine Präordnung beschrieben wird.

Lemma 4.4.6. *Die Relation $\leq_{con}$ ist reflexiv und transitiv.*

Beweis. Die Reflexivität ist sofort aus der Definition von $\leq_{con}$ ersichtlich. Für die Transitivität nehmen wir $r \leq_{con} s$ und $s \leq_{con} t$ an. Es gilt dann sowohl

$$(\forall P[\cdot])\ P[r] \Downarrow \Longrightarrow\ P[s] \Downarrow$$

als auch

$$(\forall P[\cdot])\ P[s] \Downarrow \Longrightarrow\ P[t] \Downarrow$$

wodurch mit

$$(\forall P[\cdot])\ P[r] \Downarrow \Longrightarrow\ P[s] \Downarrow\ \wedge\ P[s] \Downarrow \Longrightarrow\ P[t] \Downarrow$$

sofort die Behauptung

$$(\forall P[\cdot])\ P[r] \Downarrow \Longrightarrow\ P[t] \Downarrow$$

folgt. □

Wir werden die Definition 4.4.5 auch daraufhin untersuchen, inwieweit sie der Absicht, $(\boldsymbol{Y}\ \boldsymbol{K})$ durch Weglassen der 2. Forderung aus Abschnitt 4.4.2 mit bedeutungslosen Termen zu identifizieren, entspricht. Dazu machen wir folgende Beobachtung.

Korollar 4.4.7. *Für einen beliebigen Term $t \in \Lambda_C$ gilt $s \leq_{con} t$, sofern s*

- *nicht wohlgetypt ist,*
- *keine WHNF hat oder*
- *$s \equiv (\boldsymbol{Y}\ \boldsymbol{K})$.*

Die Terme mit den genannten Eigenschaften sind also bezügl. $\leq_{con}$ allesamt kleinste Elemente.

Beweis. Dies stellt eine unmittelbare Folgerung aus der Definition von $\leq_{con}$ und Lemma 4.4.5 dar. □

4.4.5 Kontextuelle Gleichheit

Jetzt können wir erklären, wann zwei Terme als verhaltensgleich betrachtet werden sollen.

Definition 4.4.6. Seien $s, t \in \Lambda_C$, dann gelte

$$s =_{con} t \text{ gdw. } s \leq_{con} t \;\wedge\; t \leq_{con} s$$

Für $s =_{con} t$ sagen wir auch, s und t seien *kontextuell gleich.*

Als erstes muß gezeigt werden, daß auf diese Weise auch wirklich eine Gleichheit definiert wurde, d.h. $=_{con}$ muß eine Kongruenz im Sinne der Definition 2.3.3 sein.

Lemma 4.4.8. *Die Relation $=_{con}$ ist eine Äquivalenzrelation.*

Beweis. Die Symmetrie folgt direkt aus der Definition von $=_{con}$ wie auch Reflexivität sowie Transitivität unter Anwendung von Lemma 4.4.6. $\square$

Satz 4.4.9. *Die Relation $=_{con}$ ist eine Kongruenz.*

Beweis. $=_{con}$ ist nach Lemma 4.4.5 eine Äquivalenzrelation, also müssen wir

$$s =_{con} t \implies (\forall P[\cdot])\; P[s] =_{con} P[t]$$

zeigen. Nach Definition von $=_{con}$ gilt für jeden beliebigen Kontext $P'[\cdot]$, daß

$$P'[s] \Downarrow \iff P'[t] \Downarrow$$

Nun ist für jeden beliebigen Kontext $P[\cdot]$ wiederum auch $P''[\cdot] \equiv P'[P[\cdot]]$ ein Kontext für $=_{con}$, so daß laut Definition

$$P''[s] \Downarrow \iff P''[t] \Downarrow$$

gilt und die Behauptung bewiesen ist. $\square$

Wichtig ist in diesem Zusammenhang auch, daß die durch die Reduktionsrelation $\rightarrow_\delta$ gegebene Beziehung in $=_{con}$ enthalten ist[16].

Lemma 4.4.10. *Seien $s, t \in \Lambda_C$, so gilt*

$$s \xrightarrow{*}_\delta t \implies s =_{con} t$$

Beweis. Aufgrund der Konfluenz der Reduktionsrelation nach Lemma 4.2.2 liegt es auf der Hand, daß zwei Terme gleichmäßig zu einer CWHNF konvergieren, wenn der eine auf den anderen reduziert. $\square$

[16] Dieses Ergebnis kann man in gewissem Maße als Entsprechung zu [Abr90, Proposition 2.7 (i)] sehen (vgl. auch Abschnitt 2.4.3).

Als gleich stellen sich nun sofort die Terme heraus, die bezüglich der Ordnung kleinste Elemente sind, also alle nicht wohlgetypten Terme sowie diejenigen ohne WHNF und $(\boldsymbol{Y}\ \boldsymbol{K})$.

Korollar 4.4.11. *Seien $r, s \in \Lambda_C$, sodaß $r \in NWT$ und s keine WHNF hat, dann gilt*

$$r =_{con} s =_{con} (\boldsymbol{Y}\ \boldsymbol{K})$$

Beweis. Dies ist nichts weiter als die Anwendung von Korollar 4.4.7 auf die Definition der kontextuellen Gleichheit. □

Da `bot` *per definitionem* keine WHNF hat, betrachten wir diesen Term als *Repräsentanten* seiner $=_{con}$-Äquivalenzklasse. In dieser sind generell all die Funktionen enthalten, die auf sämtlichen Argumenten undefiniert sind.

Korollar 4.4.12. *Habe $f \in \Lambda_C$ eine FWHNF $\texttt{sc}\ e_1 \dots e_i$ mit dem Superkombinator `sc`, aber für keine $e_{i+1}, \dots, e_{ar(\texttt{sc})} \in \Lambda_C$ hat $\texttt{sc}\ e_1\ \dots\ e_{ar(\texttt{sc})}$ eine WHNF, so gilt $f =_{con} \texttt{bot}$.*

Beweis. Die Aussage ist offensichtlich, da $f \Uparrow$ und $P[f]$ für irgendeinen Kontext $P[\cdot]$ nur dann zu einer CWHNF konvergiert, wenn dies auch $P[\texttt{bot}]$ tut (vgl. Lemma 4.4.4). □

Außerdem gehören zu dieser Äquivalenzklasse auch all die Terme, bei denen die Auswertung in normaler Ordnung auf den Redex `bot` trifft.

Lemma 4.4.13. *Sei R ein Reduktionskontext, dann gilt*

$$R[\texttt{bot}] =_{con} \texttt{bot}$$

Beweis. $R[\texttt{bot}]$ hat keine WHNF, da `bot` keine hat. □

Bemerkung. Insbesondere ist $R[\cdot] \equiv \texttt{case}_A\ \cdot\ e_1 \dots e_{|A|}$ ein Reduktionskontext für beliebige $e_i \in \Lambda_C, 1 \leq i \leq |A|$ und somit gilt

$$\texttt{case}_A\ \texttt{bot}\ e_1 \dots e_{|A|} =_{con} \texttt{bot}$$

Man kann die Aussage von Lemma 4.4.13 noch weiter ausdehnen auf Terme, die $=_{con}$-äquivalent zu `bot` sind.

Lemma 4.4.14. *Sei R ein Reduktionskontext und $s, t \in \Lambda_C$, so gilt*

$$s \xrightarrow[\delta]{*}{}^{no}\ R[t]\ \wedge\ t =_{con} \texttt{bot} \implies s =_{con} \texttt{bot}$$

Beweis. Nach Lemma 4.4.10 gilt $s =_{con} R[t]$. Aus $t =_{con} \texttt{bot}$ folgt $R[t] =_{con} R[\texttt{bot}]$, weil $=_{con}$ eine Kongruenz ist. Nach Lemma 4.4.13 gilt $R[\texttt{bot}] =_{con} \texttt{bot}$ und so folgt die Behauptung über die Transitivität von $=_{con}$. □

Bemerkung. Da $\cdot\ r$ für jeden Term $r \in \Lambda_C$ ein Reduktionskontext ist, gilt insbesondere

$$s =_{con} \texttt{bot} \implies (\forall r \in \Lambda_C)\ s\ r =_{con} \texttt{bot}$$

In manchen Betrachtungen werden Kontexte benötigt, die mehrere Löcher besitzen.

Definition 4.4.7. Wir definieren einen *Mehrloch-Programmkontext*, kurz *M-Kontext* oder n-stelligen Kontext induktiv[17] durch

$e \in \Lambda_C$ ist ein M-Kontext.

$[\,]$ ist ein M-Kontext.

Für beliebige M-Kontexte $C_1[\,]$ und $C_2[\,]$ ist auch $C_1[\,]\ C_2[\,]$ ein M-Kontext.

Wir möchten natürlich gerne erreichen, daß $=_{con}$ auch im Bezug auf den M-Kontextbegriff eine Kongruenz ist.

Behauptung 4.4.15. *Seien $s_i, t_i \in \Lambda_C$ für $1 \leq i \leq n$ und $C[\,]$ ein beliebiger n-stelliger Kontext, so gilt*

$$s_1 =_{con} t_1, \ldots, s_n =_{con} t_n \implies C[s_1, \ldots, s_n] =_{con} C[t_1, \ldots, t_n]$$

Für den Beweis dieser Behauptung ist es von Nutzen, einen Begriff „verhaltensgleicher Kontexte" zu bilden.

Definition 4.4.8. Wir erweitern die Relation $=_{con}$ auf (einstellige) Kontexte $P[\cdot]$ und definieren

$$P[\cdot] =_{con} P'[\cdot] \iff (\forall s \in \Lambda_C)\ P[s] =_{con} P'[s]$$

Als Kernaussage kann man dann folgenden Satz zeigen.

Satz 4.4.16. *Sind $s, s' \in \Lambda_C$ und $P[\cdot], P'[\cdot]$ zwei Programmkontexte, so gilt*

$$s =_{con} s' \ \wedge\ P[\cdot] =_{con} P'[\cdot] \implies P[s] =_{con} P'[s'] \tag{4.2}$$

Beweis. Da $=_{con}$ eine Kongruenz ist, gilt aufgrund von $s =_{con} s'$ auch $P'[s] =_{con} P'[s']$. Außerdem ist nach Definition von $P[\cdot] =_{con} P'[\cdot]$ insbesondere $P[s] =_{con} P'[s]$. Mittels der Transitivität von $=_{con}$ folgt nun

$$P[s] =_{con} P'[s] =_{con} P'[s']$$

was zu zeigen war. □

[17] Dies ist nun die eigentliche Entsprechung zu Definition 2.2.2, daher verwenden wir die Klammern $[\,]$ hier auch wieder ohne Punkt $\cdot$ darin.

Über kontextuell gleiche Ausdrücke kann jetzt eine Beziehung von 2-stelligen zu verhaltensgleichen Kontexten hergestellt werden.

Lemma 4.4.17. *Seien $s, s' \in \Lambda_C$ und $C[\,]$ ein beliebiger 2-stelliger Kontext. Setzen wir außerdem $P[\cdot] \equiv C[s, \cdot\,]$ und $P'[\cdot] \equiv C[s', \cdot\,]$, so gilt*

$$s =_{con} s' \implies P[\cdot] =_{con} P'[\cdot]$$

Beweis. Zu jedem $t \in \Lambda_C$ kann man ein $D_t[\cdot]$ finden[18], so daß $D_t[\cdot] \equiv C[\,\cdot\,, t]$. Weil $=_{con}$ eine Kongruenz ist, gilt $D[s] =_{con} D[s']$ für alle Kontexte $D[\cdot]$, also insbesondere auch für $D_t[\cdot]$ zu beliebigem $t \in \Lambda_C$. Das aber heißt nichts anderes als $(\forall t \in \Lambda_C)\ C[s,t] =_{con} C[s',t]$, was die Behauptung darstellt. □

Korollar 4.4.18. *Sei $C[\,]$ ein 2-stelliger Kontext und $s, s', t, t' \in \Lambda_C$, dann*

$$s =_{con} s' \ \wedge\ t =_{con} t' \implies C[s,t] =_{con} C[s',t']$$

Beweis. Dies folgt aus Satz 4.4.16 in Verbindung mit Lemma 4.4.17. □

Die Verallgemeinerung der Aussage von Korollar 4.4.18 auf n-stellige Kontexte erhält man durch vollständige Induktion, womit sich Behauptung 4.4.15 als gültig erweist.

Außerdem kann man die kontextuelle Gleichheit auch durch das Verhalten in bestimmten Kontexten charakterisieren.

Lemma 4.4.19. *Seien $s, t \in \Lambda_C$ Terme, die keine SCWHNF haben, und gelte $s\ r =_{con} t\ r$ für alle $r \in \Lambda_C$, so folgt daraus $s =_{con} t$.*

Beweis. Sind s und t beide $=_{con}$-äquivalent zu `bot`, so ist nichts zu zeigen. Für $s =_{con}$ `bot` folgt $s\ r =_{con}$ `bot` nach Lemma 4.4.14 und $t \neq_{con}$ `bot` stellte nun einen Widerspruch zu $s\ r =_{con} t\ r$ dar, weil t keine SCWHNF hat. Daher können wir $s \neq_{con}$ `bot` und $t \neq_{con}$ `bot` voraussetzen.

Hierfür wird die Behauptung nun durch Widerspruch bewiesen. Angenommen, es gelte $(\forall r \in \Lambda_C)\ s\ r =_{con} t\ r$ und $(\exists P[\cdot])\ P[s] \Downarrow \iff P[t] \Uparrow$. Es wird nun gezeigt, daß es einen solchen Kontext $P[\cdot]$ nicht geben kann, indem alle Möglichkeiten unterschieden werden.

1. Ist s in $P[s]$ kein notwendiger Redex, so kann $P[\cdot]$ nicht das Gegenbeispiel sein, da $P[s] \Downarrow \iff P[t] \Downarrow$ nach Lemma 4.4.4 im Widerspruch zur Annahme.
2. Also müßte s in $P[s]$ ein notwendiger Redex sein.
   (a) Ist t in $P[t]$ kein notwendiger Redex, sondern s, weil es schon in $P[\cdot]$ auftaucht, so ist dies auch nicht das Gegenbeispiel, da Lemma 4.4.4 abermals zu einem Widerspruch führte.

[18] Dies ist ohne weiteres der Fall; dabei spielt es auch keine Rolle, ob t schon in $C[\,]$ auftaucht.

(b) Also bleibt als letzte Möglichkeit, daß s in $P[s]$ und entsprechend t in $P[t]$ ein notwendiger Redex ist. Nun müssen wir anhand von s und t unterscheiden.

i. Hat s (oder t) eine ungesättigte CWHNF, so muß auch t (bzw. s) eine CWHNF mit demselben Konstruktor und derselben Anzahl von Argumenten haben, da wegen $s\ r =_{con} t\ r$ und Lemma 4.4.3 für alle Kontexte $P'[\cdot]$ gilt

$$P'[s\ r] \Downarrow_S \mathtt{con}\ s_1\ \ldots\ s_{ar(\mathtt{con})-1}\ r \iff P'[t\ r] \Downarrow_S \mathtt{con}\ t_1\ \ldots\ t_{ar(\mathtt{con})-1}\ r$$

Dies aber bedeutete

$$P'[s] \Downarrow \mathtt{con}\ s_1\ \ldots\ s_{ar(\mathtt{con})-1} \iff P'[t] \Downarrow \mathtt{con}\ t_1\ \ldots\ t_{ar(\mathtt{con})-1}$$

im Widerspruch zur Voraussetzung.

ii. Also müßten s und t beide eine FWHNF haben, die keine CWHNF ist. Dann aber könnten sie nach der Struktur eines Reduktionskontextes nur unter zwei Bedingungen notwendige Redexe in $P[s]$ resp. $P[t]$ sein:

- $P[\cdot] \equiv P'[\mathtt{case}_A\ \cdot]$ kann nicht sein, da dann $P[s], P[t] \in NWT$ im Widerspruch zur Voraussetzung.
- $P[\cdot] \equiv P'[\cdot\ r]$ für ein beliebiges r stellt einen Widerspruch zu $s\ r =_{con} t\ r$ dar.

Somit führen alle Fälle zu einem Widerspruch und die Behauptung ist bewiesen. □

Damit ist gezeigt, daß $=_{con}$ extensional auf Termen ist, die keine SCWHNF haben. Für solche, die eine SCWHNF haben, kann $=_{con}$ selbstverständlich nicht extensional sein. Denn übersättigte Konstruktoren werden stets als nicht wohlgetypt identifiziert; die Datenobjekte aber, die durch die entsprechenden SCWHNF's repräsentiert werden, können sich sehr wohl unterscheiden.

4.4.6 Wohlgetypte Terme und Gleichheit

Die Definition von $\leq_{con}$ kommt zwar ohne die Benutzung von WT aus, aber es bestehen doch Wechselwirkungen zum Begriff der Wohlgetyptheit. Zum Beispiel brachte die Art, auf die WT definiert wurde[19], es mit sich, daß alle Terme, die zu irgendeiner CWHNF konvergieren, als wohlgetypt betrachtet werden (vgl. Korollar 4.4.2).

[19] Indem nämlich der Auswertung in normaler Ordnung gefolgt wurde.

Dadurch, daß die Menge NWT der nicht wohlgetypten Terme minimal gewählt ist, gibt es viele reale Typsysteme, die strenger verfahren, d.h. nicht alle Terme aus WT ihrerseits als wohlgeytpt einordnen.

Es stellt sich nun die Frage, welche Auswirkungen die Benutzung eines solchen Typsystems auf die Gleichheitsanalyse hätte.

Das Typsystem nach Damas-Hindley-Milner

Diese Frage kann im Rahmen dieser Arbeit nur kurz erörtert werden; das soll nun anhand des Typsystems von Damas, Hindley und Milner (siehe [Hin69] oder [DM82]) geschehen, welches wir mit WT^{DHM} bezeichnen.

Alle nach dem Damas-Hindley-Milner Typsystem wohlgetypten Terme wurden bisher auch stets als wohlgetypt betrachtet. Denn in [Sch] finden wir, daß $WT^{DHM} \subsetneq WT$. Das heißt aber wiederum, daß $NWT^{DHM} \supsetneq NWT$ und so Korollar 4.4.2 nicht ohne weiteres mehr für WT^{DHM} gültig ist. Man erhält also stark abweichende Eigenschaften für $=_{con}$, legt man den Betrachtungen WT^{DHM} zugrunde, z.B. werden nicht mehr alle nicht wohlgetypten Terme untereinander und mit `bot` identifiziert.

Das ist bei der Anwendung der Implementierung der Gleichheitsanalyse zu beachten. Grundsätzlich stellt es zwar kein Problem dar, wenn strengere Kriterien[20] an das Kernsprachenprogramm eingehalten werden, als es WT fordert. Nur werden im Verlauf der Analyse (siehe Kapitel 5) auch neue Terme erzeugt. Zu untersuchen, ob oder unter welchen Bedingungen diese wiederum wohlgetypt im Damas-Hindley-Milner Typsystem sind, falls es die Eingabe war, würde hier zu weit führen.

Es sei hier nur ein Beispiel dafür angeführt, daß es $=_{con}$-Gleichheiten zwischen im Damas-Hindley-Milner Typsystem wohlgetypten und nicht wohlgetypten Termen gibt.

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

head x = case x of
  <1>      -> undefined;
  <2> x xs -> x;

dblhead x = head (head x)
```

Es ist `twice head` $\in WT \setminus WT^{DHM}$, wie man sich leicht überlegen kann. Auf der anderen Seite ist `dblhead` $\in WT^{DHM}$ und es gilt `twice head` $=_{con}$ `dblhead`, wofür der Beweis in Anhang A zu finden ist.

[20] Z.B. kann die Eingabe aus der Transformation eines Haskell-Programmes entstanden sein, welches im Sinne des Damas-Hindley-Milner Typsystems wohlgetypt sein mußte.

Kapitel 5

Die Gleichheitsanalyse und ihre Implementierung

In diesem Kapitel wird der Kalkül vorgestellt, mit dem die Gleichheit von Kernsprachenausdrücken bewiesen wird. Da dieser als Eingabe abstrakte Ausdrücke verarbeitet, besteht eine Lösung aus den möglichen Belegungen für die freien Variablen, mit denen die Ausdrücke dann $=_{con}$-gleich sind. Diese Mengen von Λ_C-Termen werden durch die in [Sch] für die Kontextanalyse verwendeten Anforderungen dargestellt.

Daher wird in Abschnitt 5.1 zuerst kurz auf den Kontextkalkül und das zugrunde liegende Prinzip von Tableaukalkülen im allgemeinen eingegangen, bevor anschließend die Struktur für das Gleichheitstableau erklärt wird. Im darauf folgenden Abschnitt werden die Regeln des Kalküls spezifiziert und begründet. Daran schließen sich die im Vorfeld der Implementierung angestellten Überlegungen an. Der nächste Abschnitt stellt dann die wesentlichen Teile der Implementierung kurz vor. Unter dem Aspekt der Arbeitsorganisation werden abschließend noch einige Bemerkungen zur Durchführung der Implementierung gemacht.

5.1 Gleichheitstableau

Wie bereits angedeutet, benutzt der Kalkül ein Tableau, um die Gleichheit von Ausdrücken zu beweisen. Die Grundlage dieser Arbeit ist in [SS96a] geschaffen worden, wo der ursprüngliche Tableaukalkül für Gleichheitsbeweise entwickelt wurde.

Dort wird auch erörtert, daß in bestimmten Fällen die Notwendigkeit besteht, Beschreibungen für Mengen von Termen vorauszusetzen, die für die freien Variablen eingesetzt werden dürfen, damit das Tableau die korrekte Lösung repräsentiert.

Diese Arbeit geht darüber hinaus, indem diejenigen Belegungen der freien Variablen, für welche die Gleichheit gilt, als Teil der Lösung *berechnet* werden. Dabei werden sowohl Techniken als auch konkrete Regeln aus dem Kontextkalkül in [Sch] benutzt, wodurch schließlich der Zusatz „unter Verwendung der Kontextanalyse“ zum Titel dieser Arbeit gerechtfertigt ist. Im folgenden werden daher die wichtigsten Prinzipien dazu erläutert.

5.1.1 Tableaukalküle und die Kontextanalyse

Die Kontextanalyse soll die Frage beantworten, für welche Belegungen von freien Variablen eines abstrakten Terms dieser durch eine gegebene Menge von Termen repräsentiert wird.

Dies wird in [Sch] mit einem Tableaukalkül gelöst. Diese Technik stammt ursprünglich aus dem Bereich des automatischen Beweisens logischer Formeln. Sie wird aber schon seit längerem erfolgreich für die statische Analyse von Programmen in funktionalen Sprachen eingesetzt, sei es im Bezug auf Striktheit (siehe [Sch94] und [SSPS95]), Terminierung (vgl. [Pan96] und [Kic96]) oder eben Auswertungskontexte (siehe [Sch] und auch [SSS97]).

Tableaus

Ein *Tableau* ist dabei ein Baum, dessen Knoten die zu beweisenden Formeln aufnehmen — genauso wie in der Logik. Für den *Tableaukalkül* müssen dann Regeln angegeben werden, die ein Tableau in ein neues überführen. Dies geschieht normalerweise dadurch, daß an die bestehenden Blätter neue angehängt werden. Das Tableau heißt *geschlossen*, wenn die Blätter nur noch aus Formeln bestehen, deren Gültigkeit bereits erwiesen ist. Das können z.B. Axiome oder irgendwelche Basisfälle sein. Ein geschlossenes Tableau stellt eine Form des mathematischen Beweises für die an der Wurzel stehende Aussage dar.

Das Verfahren führt also die zu beweisende Formel des Wurzelknotens solange auf einfachere Aussagen zurück, bis diese trivialerweise wahr sind. Um zu gewährleisten, daß wirklich ein Beweis gefunden wurde, dürfen nur Regeln bzw. Transformationen angewandt werden, die *korrekt* sind. Das bedeutet, ein Blatt darf nur dann angefügt werden, wenn durch seine Gültigkeit die des Vorgängers impliziert ist.

Ist umgekehrt gewährleistet, daß die Gültigkeit des Vorgängers die des Nachfolgers einschließt, so geht keine Lösung verloren und die Regel heißt *vollständig*. Vollständigkeit ist zwar wünschenswert, für einen korrekten Kalkül aber nicht immer möglich. Denn in bestimmten Fällen kann eine unvollständige Regel die einzige Möglichkeit sein, um an einem Tableau überhaupt neue Blätter anfügen zu können.

Kontextanalyse

In der Kontextanalyse nehmen die Knoten des Tableaus u.a. Bedingungen der Form $t \in C$ auf, wobei t ein abstrakter Ausdruck ist und C eine *Anfor-*

derung. Deren Syntax ist durch das Symbol E gemäß der folgenden Menge P von Produktionen gegeben.

Anforderungs-Ausdrücke	E	$\rightarrow$	$W \mid S \mid V \mid C \mid K \mid N$
Where-Ausdrücke	W	$\rightarrow$	$Term(E)$ `where` $T = N$
Anforderungs-Schnitte	S	$\rightarrow$	$E_1 \cap \ldots \cap E_n$
Anforderungs-Vereinigungen	V	$\rightarrow$	$\langle E_1 \ldots E_n \rangle$
Anforderungs-Konstanten	C	$\rightarrow$	`Bot` | `Fun`
Anforderungs-Konstruktoren	K	$\rightarrow$	$\texttt{con}_{A,j}\ E_1 \ldots E_n$
Anforderungs-Namen	N	$\rightarrow$	$name$
Anforderungs-Muster	T	$\rightarrow$	$\texttt{con}_{A,j}\ T_1 \ldots T_n \mid \texttt{Bot} \mid var$
Anforderungs-Definitionen	D	$\rightarrow$	$N = E$

var steht dabei für eine Variable und $name$ für einen Anforderungsnamen, der durch eine Anforderungsdefinition definiert wird. $Term(E)$ wird als Symbol einer transformierten Produktionenmenge aufgefaßt, in der alle Vorkommen von E durch $Term(E)$ ersetzt und zu der die Regeln $C \rightarrow var$ sowie $K \rightarrow\ var\ Term(E_1) \ldots Term(E_n)$ hinzugefügt wurden.

Über den Begriff der *Konkretisierung* $\gamma(C)$ wird in [Sch] erklärt, welche Terme durch eine Anforderung C beschrieben werden. Da die Anforderungsnamen rekursiv auftreten können, ist dies ein sehr komplexes Unterfangen, welches wir hier nicht weiter verfolgen können.

Es ist aber an dieser Stelle sinnvoll zu formalisieren, für welche Aussage ein *Kontexttableau* einen Beweis liefert, weil wir die Begriffe in der Gleichheitsanalyse übernehmen werden.

Allgemein ist ein Knoten eines Kontexttableaus mit einem Paar (rs, σ) markiert. Dabei ist rs eine Menge von Bedingungen $s_i \in C_i$, in der Form wie wir sie oben kennengelernt haben. σ ist ein Tupel aus Termen, die nur Variablen, Konstruktoren und `bot` enthalten. Es besteht aus sovielen Komponenten, wie freie Variablen in der Wurzel auftauchen und nimmt dort naheliegenderweise gerade diese auf. Wir werden σ auch oft auffassen als eine Substitution auf den freien Variablen in die Menge der Terme, die nur aus Variablen, Konstruktoren und `bot` bestehen. Damit jene Menge nicht jedesmal auf so umständliche Weise beschrieben werden muß, wird sie wie folgt erklärt.

Definition 5.1.1.

$$\Lambda_C^{vcb} = \{\ t \in \Lambda_C^{\#} \mid \text{in } t \text{ taucht kein Superkombinator außer } \texttt{bot} \text{ auf}\ \}$$

Im folgenden tritt die oben genannte Art einer Substitution häufig auf. Daher wird in diesem Zusammenhang eine besondere Notation eingeführt.

Notation. Mit $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ ist die Substitution $[t_1/x_1, \ldots, t_n/x_n]$ gemeint, so daß $t_i \in \Lambda_C^{vcb}$ für alle t_i mit $1 \leq i \leq n$ gilt.

Definition 5.1.2. Eine *Grundsubstitution* ist eine Substitution, die nur Grundterme für Variablen einsetzt.

Mit diesem Begriff kann man nun festlegen, was eine *Lösung* für einen Knoten sein soll.

Definition 5.1.3. Eine Grundsubstitution θ heißt *Lösung* für einen Knoten $(\{s_1 \in C_1, \ldots, s_n \in C_n\}, \sigma)$ genau dann, wenn

$$(\forall i)\ \theta s_i \in \gamma\,(C_i)$$

Die Menge aller Lösungen für den Knoten (rs, σ) wird mit $U(rs, \sigma)$ bezeichnet.

Damit erlangt man schon eine recht gute Vorstellung davon, was in einem Kontexttablau berechnet wird; schließlich sucht man ja die Lösungen für die Wurzel. Wie sich diese aus denen der einzelnen Knoten zusammensetzen soll, wird wie folgt definiert.

Definition 5.1.4. Sei $(\{t_1 \in C_1, \ldots, t_n \in C_n\}, \rho)$ die Wurzel eines Kontexttableaus T und θ eine Lösung für einen Knoten (rs, σ) von T. Dann heißt θ *Lösung der Wurzel* bzw. *des Tableaus*, wenn gilt

$$(\forall i)\ \theta\sigma t_i \in \gamma\,(C_i)$$

5.1.2 Das Tableau für Gleichheitsbeweise

Nach den Vorbereitungen im vorigen Abschnitt können wir nun das Tableau für die Gleichheitsbeweise definieren.

Definition 5.1.5. Ein *Gleichheitstableau* ist ein Kontexttableau, in dem die Bedingungen von der Art $\mathtt{EQ}(s,t) \in \langle\mathtt{True}\rangle$ mit $s, t \in \Lambda_C^{\#}$ sind.

Die Verwendung des EQ-Ausdruckes als syntaktischem Konstrukt soll weniger die Herkunft vom Kontextkalkül bezeugen, als uns in die Lage versetzen, Regeln des Kontextkalküls für ein Gleichheitstableau ohne Änderungen zu verwenden[1]. Wir schreiben $s \doteq t$ anstelle von $\mathtt{EQ}(s,t) \in \langle\mathtt{True}\rangle$, wenn wir dies nicht benötigen.

Lösungen für ein Gleichheitstableau sollen nun die kontextuelle Gleichheit gelten lassen.

Definition 5.1.6. Eine Grundsubstitution θ heißt Lösung für einen Knoten $(\{\mathtt{EQ}(s_1,t_1) \in \langle\mathtt{True}\rangle, \ldots, \mathtt{EQ}(s_n,t_n) \in \langle\mathtt{True}\rangle\}, \sigma)$ genau dann, wenn

$$(\forall i)\ \theta s_i =_{con} \theta t_i$$

gilt.

[1] Dazu verstehe sich der Begriff von Kontexten und Reduktionskontexten auch auf EQ-Ausdrücke ausgedehnt.

Es wäre zwar wünschenswert gewesen, implizit diese Beziehung zwischen den EQ-Ausdrücken und der kontextuellen Gleichheit herstellen zu können. Aber dafür wäre es nötig, auf eine geeignete Art festzulegen, wann

$$(\forall i)\ \theta\texttt{EQ}(s_i, t_i) \in \gamma\ (\langle\texttt{True}\rangle)$$

erfüllt ist, um zeigen zu können, daß dies gleichbedeutend zu

$$(\forall i)\ \theta s_i =_{con} \theta t_i$$

ist. Das ist nicht ohne weiteres möglich, da EQ nicht durch eine Funktion definiert werden kann. Denn dazu wären Reduktionsregeln erforderlich, die EQ(bot, bot) zu True reduzieren ließen, EQ(con, bot) bzw. EQ(bot, con) hingegen zu False. Deswegen wurde die direkte Definition durch 5.1.6 vorgezogen.

5.2 Kalkülregeln

Wie schon angekündigt, wird es Regeln geben, die ein Gleichheitstableau in ein anderes überführen.

Zum einen wird dabei ein Tableau durch Anfügen neuer Blätter erweitert. Dies geschieht allein aufgrund lokaler Informationen, die der Vorgänger bereitstellt. Daher nennen wir solche Regeln *Expansionsregeln.*

Darüberhinaus gibt es welche, die nicht-lokale, möglicherweise über das gesamte Tableau verstreute Informationen benötigen. Solche Regeln heißen *Meta-Regeln.*

Im folgenden werden zuerst die Expansionsregeln und danach die Meta-Regeln des Kalküls behandelt. Dabei werden jeweils auch gleich Betrachtungen zu Korrektheit und Vollständigkeit angestellt.

5.2.1 Expansionsregeln

Die Expansionsregeln werden in folgender Form notiert:

$$\frac{\text{Altes Blatt}}{\text{Neues Blatt}_1 \mid \ldots \mid \text{Neues Blatt}_n}, \text{ falls Bedingung}$$

„Altes Blatt“ ist die Beschriftung des bestehenden Blattes im Tableau, an welches die Blätter mit den Beschriftungen „Neues Blatt_1“ bis „Neues Blatt_n“ angefügt werden können. Diese stellen mehrere mögliche Alternativen dar, die daher durch | getrennt werden. „Bedingung“ legt fest, unter welchen Voraussetzungen die Regel angewendet werden darf. In den Knoten bzw. Blättern[2] werden alle Klammern weggelassen, d.h. $s \doteq t, R, \sigma$

[2] Genauer gesagt: in deren Beschriftungen

steht für $(\{s \doteq t, R\}, \sigma)$, wenn $R = R_1, \ldots, R_m$ eine Reihe von m weiteren Bedingungen ist.

Bekanntlich ist $=_{con}$ symmetrisch und wegen Definition 5.1.6 gilt dies auch für die Formeln bzw. im Bezug auf deren Lösungen, Das heißt, eine Grundsubstitution θ ist eine Lösung für die Formel $s \doteq t$ genau dann, wenn sie auch eine Lösung für $t \doteq s$ ist.

Einige der Expansionsregeln erfordern für die linke und/oder rechte Seite einer Formel eine bestimmte Form. Wegen der o.g. Symmetrie genügt es, eine Regel nur für einen der beiden Fälle anzugeben. Im folgenden sei die Regel für den symmetrischen Fall dadurch stets implizit spezifiziert.

Reduktion

Jede Geichheitsformel enthält je zwei Ausdrücke $s, t \in \Lambda_C^{\#}$. Mit Hilfe der in Abschnitt 4.3 definierten abstrakten Reduktion kann nun jeder dieser beiden reduziert werden, und zwar geschieht das in normaler Ordnung.

$$\text{(AbsRed)} \qquad \frac{s \doteq t, R, \sigma}{s' \doteq t, R, \sigma \mid s \doteq t', R, \sigma}, \text{ falls } s \rightarrow_{\delta}^{\text{no}} s' \text{ bzw. } t \rightarrow_{\delta}^{\text{no}} t'$$

Wie bereits angesprochen genügt es wegen der Symmetrie von $=_{con}$, für die Korrektheit dieser Regel die Reduktion auf einer Seite, o.B.d.A. der linken, zu betrachten. Ist θ eine beliebige Lösung für $s' \doteq t$ und gelte $s \rightarrow_{\delta}^{\text{no}} s'$, so muß folgende Bedingung erfüllt sein.

$$\theta s' =_{con} \theta t \implies \theta s =_{con} \theta t \tag{5.1}$$

Aufgrund von Lemma 4.4.10 kann aus $\theta s \rightarrow_{\delta} \theta s'$ gefolgert werden $\theta s =_{con} \theta s'$. Damit die Behauptung also über die Transitivität von $=_{con}$ folgt, muß

$$s \rightarrow_{\delta} s' \implies \theta s \rightarrow_{\delta} \theta s'$$

gezeigt werden. Ist s ein Grundterm, so auch s' und es ist klar, daß $\theta s \rightarrow_{\delta} \theta s'$, da durch θ keine Variablen ersetzt werden. Ist s ein abstrakter Term, so erfolgt aber durch den Schritt $s \rightarrow_{\delta} s'$ keine besondere Behandlung der Variablen, weshalb $\theta s \rightarrow_{\delta} \theta s'$ wiederum erfüllt ist. Dieses Vorgehen ist auch für die umgekehrte Richtung der Implikation zulässig, womit sogar

$$\theta s' =_{con} \theta t \iff \theta s =_{con} \theta t$$

gilt und wir nebenbei die Vollständigkeit der Regel erhalten.

Hinzufügen von Argumenten

Wenn linke und rechte Seite der Formel FWHNF's sind, können die fehlenden Argumente hinzugefügt werden.

$$\text{(AddArgs)} \qquad \frac{f \doteq g, R, \sigma}{f\ x \doteq g\ x, R, \sigma}, \text{ falls } f \text{ und } g \text{ in FWHNF}$$

Hierbei muß x eine neue Variable sein, die ursprünglich weder in f noch in g auftauchte, also $x \notin \mathcal{FV}(f) \cup \mathcal{FV}(g)$. Die Vollständigkeit dieser Regel fällt sofort ins Auge, da $=_{con}$ eine Kongruenz ist. Denn für jede Lösung θ von $f \doteq g$ gilt

$$\theta f =_{con} \theta g \implies \theta P_x[f] =_{con} \theta P_x[g]$$

für alle Kontexte $P_x[\cdot] \equiv \cdot\, x$ mit beliebigem $x \in \Lambda_C$. Genauer müßte man sagen, daß jede Lösung θ von $f \doteq g$ auf $f\ x \doteq g\ x$ eine ganze Lösungsschar hervorbringt, so daß es dort für jedes $t \in \Lambda_C$ eine Lösung θ_t gibt mit

$$\theta_t s = \begin{cases} t & \text{falls } s \equiv x \\ \theta s & \text{sonst} \end{cases}$$

Somit bedeutet die Korrektheit bei dieser Regel nicht nur, daß

$$\theta f\ x =_{con} \theta g\ x \implies \theta f =_{con} \theta g$$

für jede Lösung θ von $f\ x \doteq g\ x$ gelten muß. Denn dies ist trivialerweise der Fall, da eine Lösung für $f\ x \doteq g\ x$ die Variable x ersetzt, die in $f \doteq g$ gar nicht auftaucht. Damit die Voraussetzungen für Lemma 4.4.19 erfüllt sind, muß vielmehr auch

$$\begin{aligned} (5.2)\quad \theta \in U(f\ x \doteq g\ x, \sigma) \implies \\ (\forall\, t \in \Lambda_C)\ (\exists \theta_t \in U(f\ x \doteq g\ x, \sigma)) \\ (s \not\equiv x \implies \theta s = \theta_t s)\ \wedge\ \theta_t x = t \end{aligned}$$

gefordert werden. Hierdurch wird verdeutlicht, daß x eine allquantifizierte Variable ist, an die keinerlei Einschränkungen erlaubt sind. Um dies sicherzustellen, müssen die Variablen, die durch die Regel (AddArgs) hinzugefügt werden, streng von den übrigen unterschieden werden. Dazu scheint ein besonderes Augenmerk bei der Definition aller anderen Kalkülregeln vonnöten. Glücklicherweise ist das nicht der Fall: Die Forderung (5.2) wurde durch die vorige Regel (AbsRed) nicht berührt und muß generell nur bei denen mit einbezogen werden, die neue Belegungen für Variable einführen. Das geschieht z.B. bei der unmittelbar folgenden Regel (CaseFV).

case auf freien Variablen

Eine Anwendung von case auf eine freie Variable kann nicht weiter reduziert werden, sondern stellt u.U. einen *potentiellen* Redex dar.

Definition 5.2.1. Ein Ausdruck $s \in \Lambda_C$ heißt *potentieller Redex* in $P[s]$, falls es eine Substitution σ gibt, so daß σs ein Redex in $\sigma P[s]$ ist.

In einer solchen Anwendung des `case`, können alle Konstruktorausdrücke, die vom Typ her zulässig sind, und `bot` eingesetzt werden. In den folgenden Betrachtungen steht $D[\cdot]$ also für einen Reduktionskontext.

(CaseFV) $$\frac{D[\texttt{case}_A\ x\ e_1\ \ldots\ e_{|A|}] \in\ \langle\texttt{True}\rangle, R, \sigma}{L_0 \mid\ \ldots\ \mid L_{|A|}},$$ falls x nicht durch (AddArgs) eingeführt

Hier sind die neuen Blätter durch

$$L_i = \rho_i(D[\texttt{case}_A\ x\ e_1\ \ldots\ e_{|A|}] \in\ \langle\texttt{True}\rangle, R), \rho_i\ \circ\ \sigma$$

gegeben, wobei $\rho_0 = \{x \mapsto \texttt{bot}\}$ und $\rho_i = \{x \mapsto \texttt{con}_{A,i}\ x_{i,1} \ldots x_{i,ar(\texttt{con}_{A,i})}\}$ für $0 < i \leq |A|$ ist. Die $x_{i,j}$ müssen neue Variablen sein.

Für die Korrektheit dieser Regel muß gezeigt werden, daß $\theta\ \circ\ \rho_i$ eine Lösung für $D[\texttt{case}_A\ x\ e_1\ \ldots\ e_{|A|}] \in \langle\texttt{True}\rangle$ ist, falls θ eine Lösung für eines der L_i ist. Dies ist offensichtlich der Fall, da die L_i gerade durch die Anwendung der ρ_i aus dem ursprünglichen Blatt hervorgehen.

Die Festlegung, daß x nicht durch die Regel (AddArgs) eingeführt worden sein darf[3], ist für die Bedingung (5.2) notwendig.

Regeln für `bot`

Da $=_{con}$ eine Kongruenz ist, könnten ganz allgemein kontextuell gleiche Ausdrücke in den Formeln gegeneinander ausgetauscht werden. Diese Gleichheit aber soll die Analyse gerade beweisen, sodaß eine Ersetzung nur in den Fällen sinnvoll ist, in denen sie offenkundig ist.

Dies trifft zum Beispiel auf Ausdrücke zu, in denen `bot` ein notwendiger Redex ist. Wird dies festgestellt, kann der gesamte Ausdruck durch `bot` ersetzt werden.

$$\frac{D[\texttt{bot}] \doteq t, R, \sigma}{\texttt{bot} \doteq t, R, \sigma}, \text{ falls } D[\cdot] \text{ ein Reduktionskontext}$$

Diese Regel ist wegen Lemma 4.4.13 korrekt und vollständig. Anwenden wird man sie häufig in der speziellen Instanz

(CaseBot) $$\frac{D[\texttt{case}_A\ \texttt{bot}] \in\ \langle\texttt{True}\rangle, R, \sigma}{D[\texttt{bot}] \in\ \langle\texttt{True}\rangle, R, \sigma}$$

für einen Reduktionskontext $D[\cdot]$, da solche $\texttt{case}_A$-Ausdrücke insbesondere durch die Anwendung der Regel (CaseFV) entstehen.

[3] Auf den ersten Blick ist dies scheinbar keine lokale Information mehr; sie kann aber ohne weiteres in den Knoten des Tableaus mitgeführt werden.

Da bot keine WHNF hat, gibt es offensichtlich keine Lösung für die Gleichheit zu einer CWHNF. Dies wird deutlich gemacht, indem eine weitere Marke *no!* an das Blatt geschrieben wird.

$$\frac{\texttt{con}\ s_1\ \ldots\ s_n \doteq \texttt{bot}, R, \sigma}{\textit{no!}\ \texttt{con}\ s_1\ \ldots\ s_n \doteq \texttt{bot}, R, \sigma}$$

Nach den obigen Ausführungen ist diese Regel korrekt und vollständig.

Approximation

Tritt eine freie Variable an der Stelle eines Funktionssymbols auf, so bezeichnen wir das wie folgt.

Definition 5.2.2. Ein Ausdruck $f\ e_1\ \ldots\ e_n$ heißt *abstrakte Applikation*, wenn f eine Variable ist.

Eine abstrakte Applikation stellt einen potentiellen Redex dar. Man kann aber nicht ohne weiteres angeben, wie die freie Variable zu ersetzen wäre, um sinnvoll fortzufahren. Denn sie kann für alle möglichen Konstruktoren oder definierte Superkombinatoren stehen oder sogar für Funktionen, die nur bestimmte Eigenschaften erfüllen müssen.

Um mehr über diese Eigenschaften herauszufinden, kann eine abstrakte Applikation *approximiert* werden, wenn sie im Kontext eines case auftaucht. Ein solcher case-Ausdruck stellt nämlich wiederum einen potentiellen Redex dar. Die abstrakte Applikation wird in einem case-Kontext also durch eine neue Variable ersetzt.

$$\text{(Approx)}\quad \frac{D[\texttt{case}_A\ t\ e_1\ \ldots\ e_{|A|}] \in\ \langle\texttt{True}\rangle, R, \sigma}{\rho(D[\texttt{case}_A\ x\ e_1\ \ldots\ e_{|A|}] \in\ \langle\texttt{True}\rangle, R), (\rho\ \circ\ \sigma)},\quad \text{falls } t \text{ eine abstrakte Applikation}$$

wobei $\rho = \{t \mapsto x\}$ für eine neue Variable x gesetzt ist. Die Korrektheit ist offensichtlich, da für eine Lösung θ am Blatt, $\theta\ \circ\ \rho$ eine Lösung des Vorgängers ist.

Konstruktoren

Stellen beide Seiten eine CWHNF dar, können zwei Fälle unterschieden werden. Bei gleichen Konstruktoren und gleicher Anzahl Argumente können wir dekomponieren:

$$\text{(CDecomp)}\quad \frac{\texttt{con}\ s_1\ \ldots\ s_n \doteq \texttt{con}\ t_1\ \ldots\ t_n, R, \sigma}{s_1 \doteq t_1, \ldots, s_n \doteq t_n, R, \sigma},\quad \text{falls } n \leq ar(\texttt{con})$$

Diese Regel heißt *Konstruktordekomposition* und ist korrekt, da $=_{con}$ nach 4.4.15 eine Kongruenz auf M-Kontexten ist. Wir können nun einfach den n-stelligen Kontext $C[\,] \equiv \texttt{con}\ [\,]\ \ldots\ [\,]$ angeben, so daß mit $C[s_1, \ldots, s_n] =_{con} C[t_1, \ldots, t_n]$ die Behauptung folgt.

Der Beweis der Vollständigkeit dieser Regel kommt ohne die Benutzung von Mehrloch-Kontexten aus. Denn aus

$$\mathtt{con}_{A,j}\ s_1\ \dots\ s_n\ =_{con}\ \mathtt{con}_{A,j}\ t_1\ \dots\ t_n$$

folgt für alle Kontexte $P[\cdot]$ auch

$$P[\mathtt{con}_{A,j}\ s_1\ \dots\ s_n]\ =_{con}\ P[\mathtt{con}_{A,j}\ t_1\ \dots\ t_n]$$

da $=_{con}$ eine Kongruenz ist. Dies ist insbesondere für diejenigen Kontexte $P_i[\cdot]$ zutreffend, die das i-te Argument des Konstruktors $\mathtt{con}_{A,j}$ extrahieren. Einen solchen kann man wie folgt angeben

$$P_i[\cdot] \equiv \mathtt{case}_A\ (\cdot\ x_{n+1}\ \dots\ x_{ar(\mathtt{con}_{A,j})})\ e_{i1}\ \dots\ e_{ij}\ \dots\ e_{i|A|}$$

Wobei e_{ij} die Projektion auf das i-te Element liefert, z.B. als Superkombinator definiert durch

$$\mathtt{e_{ij}}\ x_1\ \dots\ x_{ar(\mathtt{con}_{A,j})} = x_i$$

Falls hingegen die Konstruktoren nicht übereinstimmen oder die Anzahl der Argumente unterschiedlich ist, kann es keine Lösung geben.

$$\frac{\mathtt{con}_A\ s_1\ \dots\ s_m \doteq \mathtt{con}_B\ t_1\ \dots\ t_n, R, \sigma}{\mathit{no!}\ \mathtt{con}_A\ s_1\ \dots\ s_m \doteq \mathtt{con}_B\ t_1\ \dots\ t_n, R, \sigma},\quad \begin{array}{l}\text{falls } A = B \implies \\ n < m \leq ar(\mathtt{con}_A)\end{array}$$

Da der Fall übersättigter Konstruktoren gerade ausgeschlossen wird, ist die Regel korrekt und vollständig[4].

Funktionen

Stellen beide Ausdrücke eine Anwendung ein und derselben Funktion auf dieselbe Anzahl von Argumenten dar, so können wir auch hier dekomponieren.

(FDecomp) $$\frac{f\ s_1\ \dots\ s_n \doteq f\ t_1\ \dots\ t_n, R, \sigma}{s_1 \doteq t_1, \dots, s_n \doteq t_n, R, \sigma},\quad \text{falls } n \leq ar(f)$$

Dabei spielt es keine Rolle, ob f ein Superkombinator oder eine freie Variable ist. Denn die Korrektheit kann in jedem Fall genauso wie für die Konstruktoren gezeigt werden. Die Vollständigkeit bleibt im Gegensatz dazu aber nicht unbedingt gewahrt. Denn eine Funktion liefert eben *nicht nur dann* dasselbe Ergebnis, wenn sie auf dieselben Argumente angewendet wird. Das kann man sich schon an einem einfachen Gegenbeispiel wie $\mathtt{K}\ x\ y = x$ klarmachen, da dort y für das Ergebnis überhaupt keine Rolle spielt.

[4] Die Annahme $n < m$ erfolgt wegen der Symmetrie der Formeln o.B.d.A.

Syntaktische Gleichheit

Zwei syntaktisch gleiche Terme stellen den trivialen Fall dar, also kann die entsprechende Formel gestrichen werden.

$$\text{(SyntEQ)} \qquad \frac{s \doteq t, R, \sigma}{R, \sigma}, \text{ falls } s \equiv t$$

Diese Regel ist korrekt und vollständig, weil $=_{con}$ reflexiv ist.

Binden von Variablen

Die beiden Ausdrücke in einer Formel sind sicherlich gleich, wenn sie unifiziert werden können, d.h. durch eine Substitution auf den Variablen in Deckung zu bringen sind. Eine Unifikation verliefe dabei über die Struktur der Terme. Die dazu erforderlichen Schritte sind bereits durch Konstruktor- bzw. Funktionsdekomposition gegeben, sodaß folgende Regel genügt

$$\text{(BindFV)} \quad \frac{C[x] \doteq C[t], R, \sigma}{C[t] \doteq C[t], R, \{x \to t\} \circ \sigma}, \quad \begin{array}{l}\text{falls } x \in \mathcal{FV}(C[x]) \text{ nicht} \\ \text{durch (AddArgs) eingeführt}\end{array}$$

Korrekt ist die Regel, gerade weil für jede Lösung θ für $C[t] =_{con} C[t]$ die Grundsubstitution $\theta \circ \{x \mapsto t\}$ wiederum eine Lösung für $C[x] =_{con} C[t]$ ist. Wie man sieht, ist es hier wegen der Forderung (5.2) wiederum notwendig, hinzugefügte Argumente auszuschließen.

Typfehler

Da die kontextuelle Gleichheit die nicht wohlgetypten Terme miteinander identifiziert, ist folgende Regel korrekt und vollständig.

$$\frac{s \doteq t, R, \sigma}{R, \sigma}, \text{ falls } s, t \in DNWT$$

Die Einschränkung auf $DNWT$ ist wichtig, da schon NWT nicht mehr entscheidbar ist[5]. Auf der anderen Seite wissen wir aus Korollar 4.4.2, daß eine CWHNF immer wohlgetypt ist. Das rechtfertigt die folgende Abbruchregel.

$$\frac{s \doteq t, R, \sigma}{no!\ s \doteq t, R, \sigma}, \text{ falls } s \in DNWT \text{ und } t \text{ eine CWHNF ist}$$

Auch diese ist offensichtlicherweise korrekt und vollständig.

[5] Durch den Gebrauch der Reduktionsrelation: Denn schon ein Ausdruck s, der auf $t \in DNWT$ *reduziert*, ist nicht wohlgetypt. Das aber kann man nicht entscheiden; die Reduktion auf das Halteproblem liegt auf der Hand.

5.2.2 Meta-Regeln

Die Überschrift ist vielleicht ein wenig irreführend, da es sich im Prinzip nur um eine einzige Meta-Regel handelt. Auf der anderen Seite wird sie dadurch gerechtfertigt, daß es zum Teil verschiedene Varianten für die Anwendung dieser Regel gibt. Denn daher ist auch die Auffassung vertretbar, es handele sich um mehrere einzelne Meta-Regeln.

Matching

Das Kriterium, welches stets gefordert wird, ist eine gewisse Übereinstimmung zwischen zwei Knoten des Tableaus. Damit sind selbstverständlich solche Knoten gemeint, zwischen denen ein Pfad im Tableau existiert. Das soll jetzt formalisiert werden.

Definition 5.2.3. Sei $L = D[x_1, \dots, x_n] \in \langle \texttt{True} \rangle, R, \rho$ ein Knoten eines Gleichheitstableaus, in welchem die x_i sämtlich Variablen sind. Sei nun weiterhin $L' = D[t_1, \dots, t_n] \in \langle \texttt{True} \rangle, R, \sigma$ ein Nachfolger von L. Bestehen die Ausdrücke t_i nur aus Variablen, Konstruktoren und `bot`, dann sprechen wir von einem *Match*[6] zwischen L und L'.

Man kann sich das ungefähr so vorstellen, daß das Tableau quasi immer wieder um diesen bzw. ähnliche Abschnitte expandiert werden könnte. Weil man den Prozess dabei immer wieder in dem oberen Knoten L beginnen könnte, spricht man auch von einer *Schleife*. Die Lösung einer solchen Schleife wird durch einen `where`-Ausdruck beschrieben.

Die Schleifenregel

Im folgenden bezeichne T ein Gleichheitstableau mit der Wurzel $W = s \doteq t, (x_1, \dots, x_r)$. Seien L und L' wie in der Definition 5.2.3 zwei Knoten in T, d.h. es ist ein Match zwischen L und L' gegeben, dann kann das Tableau folgendermaßen um ein neues Blatt[7] erweitert werden.

$$\text{(Loop)} \qquad \frac{D[t_1, \dots, t_n] \in \langle \texttt{True} \rangle, R, \sigma}{\sigma \;\texttt{where}\; (t_1, \dots, t_n, \sigma x_{n+1}, \dots, \sigma x_r) = N, R, \sigma}$$

Dabei ist N der Name für die Lösungsanforderung der Wurzel[8], woran man erkennt, daß es sich hier um eine rekursive Definition handelt.

[6] engl. *to match* – gleichkommen, zusammenpassen

[7] Streng genommen ist die Verwendung des `where`-Ausdruckes in dem neuen Blatt nicht erlaubt, da die Bedingungen in den Knoten eines Gleichheitstableaus stets die Form $\texttt{EQ}(s,t) \in \langle \texttt{True} \rangle$ haben sollen. Aber in die Knoten eine zusätzliche Komponente alleine dazu einzuführen, an dieser Stelle den `where`-Ausdruck aufzunehmen, bedeutete einen übertriebenen Formalismus. Denn sinnvollerweise wird an einem Blatt ohnehin entweder expandiert oder eine Schleife gebildet.

[8] Dieser Begriff wird erst in Abschnitt 5.3.1 erklärt werden; siehe dazu auch Fußnote 11 auf Seite 69.

Informal soll diese folgendes aussagen: Mit $\sigma = (\sigma x_1, \ldots, \sigma x_r)$ ist eine Belegung für die Variablen der Wurzel gefunden worden, so daß die Gleichheit gilt. Variablen, die ihrerseits darin auftauchen, betrachtet man, als seien sie durch eine Anforderung der Art $(t_1, \ldots, t_n, \sigma x_{n+1}, \ldots, \sigma x_r) \in N$ eingeschränkt.

Korrektheit

Vernachlässigt wurden bisher die Voraussetzungen, unter denen eine Schleife überhaupt gebildet werden darf, damit die angegebene Lösung korrekt ist. Dies ist nämlich keineswegs immer der Fall. Vielmehr muß auf den Pfad zwischen L und L' mindestens eine der folgenden Eigenschaften zutreffen:

- Es findet eine Konstruktordekomposition statt.
- Es werden Argumente hinzugefügt.
- Es wird jeweils einmal links und rechts abstrakt reduziert.

Daß die Schleifenregel unter den obigen Bedingungen die korrekte Lösung liefert, wird in [SS96a] bzw. [Sch] ausführlich behandelt. Für den Beweis sei daher auf ebendort verwiesen. An dieser Stelle kann nur eine kurze anschauliche Erklärung geliefert werden: Die Kriterien sorgen im Prinzip dafür, daß eine Schleife nur endlich oft durchlaufen werden kann, indem die Terme in den Formeln im Bezug auf eine wohlfundierte Ordnungsrelation „kleiner“ werden. Dies liefert schließlich die Verankerung für eine Induktion.

5.3 Menge der Lösungen

Sei nun ein Gleichheitstableau T mit der Wurzel $s \doteq t, (x_1, \ldots, x_n)$ gegeben, so stellt sich die Frage, für welche Belegungen von $x_1, \ldots, x_n$ mit Grundtermen die kontextuelle Gleichheit zwischen s und t gilt. Wir werden sehen, daß sich diese Belegungen aus den Blättern ergeben, die als *gelöst* gelten dürfen.

Definition 5.3.1. Ein Blatt (rs, σ) heißt *gelöst*, wenn es keine Gleichheitsformel mehr enthält, also rs entweder leer ist oder nur noch aus den `where`-Ausdrücken von Schleifen besteht. Ein Tableau T, in welchem sämtliche Blätter gelöst sind, wird als *geschlossen* bezeichnet.

Aus der Korrektheit der Kalkülregeln folgt, daß jede Lösung eines Blattes auch eine Lösung der Wurzel nach Definition 5.1.4 ist. Für jedes Blatt ohne `where`-Ausdrücke ist also mit dem Tupel σ eine Belegung für die Variablen der Wurzel gefunden, so daß die kontextuelle Gleichheit dort für jede Grundsubstitution gilt, sofern in σ noch weitere Variablen auftauchen. Für die `where`-Ausdrücke in gelösten Blättern gilt sinngemäß das gleiche; denn auch diese repräsentieren mögliche Belegungen dafür, daß die Gleichheit an der Wurzel gilt.

5.3.1 Konstruktion der Lösungsanforderung

Um nun die gesamte Menge aller möglichen Belegungen für die Variablen der Wurzel angeben zu können, müssen daher nur die jeweiligen σ-Tupel bzw. `where`-Ausdrücke der gelösten Blätter aufgesammelt werden.

Dies geschieht in einem Anforderungsterm, wobei ein Auftreten des Superkombinators `bot` durch den Anforderungsterm `Bot` dargestellt wird. Die Repräsentierung von `Bot` ist nämlich gerade die Menge aller Terme ohne WHNF (vgl. [Sch]). Außerdem werden die ungebundenen Variablen[9] durch den Anforderungsnamen `Top` ersetzt, welcher für eine Anforderung steht, dessen Konkretisierung alle möglichen Grundterme enthält. Bei Verzweigungen, die aufgrund der Anwendung der Regel (CaseFV) entstehen, kann man einfach entsprechende Anforderungs-Vereinigungen bilden.

Das Auftreten von Approximationen

Einzig auf die Behandlung von Variablen, die durch eine Approximation neu eingeführt wurden, muß ein besonderes Augenmerk gelegt werden. Ergeben sich nämlich an eine solche Variable Einschränkungen, d.h. wird sie ihrerseits durch Terme aus Λ_C^{vcb} substituiert, so betrifft dies an sich den Ausdruck, der approximiert wurde.

Zum Beispiel erhalten wir bei einer Approximation eines Termes t durch die Variable x die Bedingung $t \in D_e$, falls x durch $e \in \Lambda_C^{vcb}$ substituiert worden ist und D_e die zu e gehörende[10] Anforderung darstellt. Solche Bedingungen können nicht weiter bearbeitet werden, da es sich bei dem approximierten Term t stets um eine abstrakte Applikation handeln muß. Sie bleiben also übrig, bilden im gewissen Sinne eine Menge von „Residuen", die u.U. mit der Kontextanalyse weiter untersucht werden könnten, wenn für die Variable, die in t an der Stelle des Funktionssymbols steht, ein definierter Superkombinator eingesetzt würde.

Da dieser Sachverhalt mit den bisherigen Mitteln nicht mehr dargestellt werden kann, wird ein neues syntaktisches Konstrukt für Anforderungs-Ausdrücke eingeführt:

$$\begin{array}{lll} \text{Anforderungs-Ausdrücke} & E & \rightarrow \ \ldots \mid B \\ \vdots & & \\ \text{Needs-Ausdrücke} & B & \rightarrow \ Term(E)\ \texttt{needs}\ t \ \in \ E \end{array}$$

[9] Bei einem `where`-Ausdruck der Art X `where` $T = N$ handelt es sich dabei um die Variablen, die höchstens einmal in X und gar nicht in T auftauchen. In einem Tupel lassen sich hingegen alle Variablen so bezeichnen, die nicht mehrfach auftreten.

[10] also im wesentlichen den Term e, in dem — wie oben schon beschrieben — alle Vorkommen von `bot` durch `Bot` und alle Variablen durch `Top` ersetzt wurden.

Dessen Bedeutung kann hier nur informal erklärt werden: C **needs** $t \in D$ soll für die Teilmenge von Termen aus $\gamma(C)$ stehen, für die die zusätzliche Bedingung $t \in D$ gilt, d.h. in C auftauchende Variablen dürfen nur Belegungen annehmen, die durch die Lösung von $t \in D$ abgedeckt werden.

Die Lösungsanforderung und die Lösungsfunktion

Damit sind nun die Voraussetzungen geschaffen, um die zu einem Gleichheitstableau gehörende *Lösungsanforderung* zu beschreiben.

Definition 5.3.2. Sei T ein Gleichheitstableau mit der Wurzel $s \doteq t, (x_1, \ldots, x_n)$, dann wird T eine *Lösungsanforderung* N zugeordnet[11], die durch

$$N = Lsg(s \doteq t, (x_1, \ldots, x_n))$$

mittels der folgenden Lösungsfunktion $Lsg(\cdot)$ definiert ist.

$$Lsg(V) = \begin{cases} \sigma & \text{falls } V = (\emptyset, \sigma) \text{ gelöst,} \\ X \;\mathtt{where}\; T \;=\; N & \text{falls } V = (X \;\mathtt{where}\; T \;=\; N, \sigma), \\ Lsg(V') \;\mathtt{needs}\; t \;\in\; D_x & \text{falls } V \text{ eine Approximation von } t \\ & \text{mit } x \text{ und } V' \text{ Nachfolger von } V, \\ \langle Lsg(V_1), \ldots, Lsg(V_n) \rangle & \text{falls } V_1, \ldots, V_n \text{ Nachfolger von } V, \\ \langle\rangle & \text{sonst (keine Lösung)} \end{cases}$$

Hierbei soll D_x für eine Anforderung stehen, die die Einschränkungen, die für die neue Variable x gefunden wurden, repräsentiert.

Man könnte die Konstruktion dieses Anforderungstermes D_x formalisieren, indem man den Knoten als eine weitere Komponente ähnlich zu dem Tupel σ eine Liste von Variablen, die durch Approximationen eingeführt wurden, hinzufügt. Bei Anwendung der Regel (Approx) wird die neue Variable an diese Liste angehängt und bei Anwendung der Regeln (CaseFV) sowie (BindFV) die entsprechende Substitution auch auf diese Liste angewendet.

Wie man sieht, ist diese Formalisierung nicht schwer, hätte die Darstellung der Kalkülregeln in Abschnitt 5.2 aber unnötig verkompliziert, weshalb sie unterblieben ist.

[11] Hierbei soll N derselbe Name sein, der auch in den `where`-Ausdrücken der Schleifen verwendet wird.

5.3.2 Gleichheitsbeweis

Nun kann schließlich genau spezifiziert werden, für welche Aussage ein Gleichheitstableau einen Beweis liefert.

Sei also T ein Gleichheitstableau mit der Wurzel $s \doteq t, (x_1, \ldots, x_n)$ und N die nach Definition 5.3.2 dazugehörende Lösungsanforderung, dann gilt

$$(\forall (e_1, \ldots, e_n) \in \gamma\,(N))\ s[e_1/x_1, \ldots, e_n/x_n] =_{con} t[e_1/x_1, \ldots, e_n/x_n]$$

aufgrund der Korrektheit der Kalkülregeln. Um das Vorgehen des Kalküls zu veranschaulichen, wird nun ein etwas umfangreicheres Beispiel gegeben.

Beispiel 5.3.1. In dem folgenden Kernsprachenprogramm bezeichne l den algebraischen Datentypen für Listen mit dem 0-stelligen Konstruktor `Nil` und dem 2-stelligen Konstruktor `Cons`.

$$\begin{array}{ll}
\texttt{f}\ xs\ y\ ys\ = & \texttt{Cons}\ y\ (\texttt{append}\ xs\ ys);\\
\texttt{g}\ xs\ y\ ys\ = & \texttt{Cons}\ y\ xs;\\
\texttt{append}\ xs\ ys\ = & \texttt{case}_l\ xs\ ys\ (\texttt{consappend}\ ys);\\
\texttt{consappend}\ ys\ x\ xs\ = & \texttt{Cons}\ x\ (\texttt{append}\ xs\ ys);
\end{array}$$

In Abbildung 5.1 ist das Tableau für die Anfrage

$$\texttt{case}_l\ ys'\ \texttt{Nil}\ (\texttt{f}\ xs) \doteq \texttt{case}_l\ ys'\ \texttt{Nil}\ (\texttt{g}\ xs)$$

zu finden. Da es sehr groß ist, wurden einige Zwischenschritte, meist Reduktionen, fortgelassen. Außerdem wurden zur besseren Übersicht jeweils vor den `where`-Ausdrücken von Schleifen und den verschiedenen Alternativen der Regel (CaseFV) horizontale Linien eingefügt. Die Lösungsanforderung zu dem Tableau aus Abbildung 5.1 wird mit *AppBsp* bezeichnet und ergibt sich zu

$$\begin{array}{rcl}
\mathit{AppBsp} & = & \langle(\texttt{Top},\texttt{Bot}),(\texttt{Top},\texttt{Nil}),\\
& & (\texttt{Bot},\texttt{Cons Top Top}),(\texttt{Nil},\texttt{Cons Top Nil}),\\
& & (\texttt{Cons Top}\ zs,\texttt{Cons}\ y\ ys)\ \texttt{where}\ (zs,\texttt{Cons}\ y\ ys)\ =\ \mathit{AppBsp}\rangle
\end{array}$$

Das bedeutet nun

$$(\forall (xs, ys') \in \gamma\,(\mathit{AppBsp}))\ \texttt{case}_l\ ys'\ \texttt{Nil}\ (\texttt{f}\ xs) =_{con} \texttt{case}_l\ ys'\ \texttt{Nil}\ (\texttt{g}\ xs)$$

5.4 Softwaredesign

Vor jeder Implementierung müssen Überlegungen zum Design der Software angestellt werden. Dies betrifft die grundsätzlichen Funktionalitäten und die dafür verwendeten Datenstrukturen genauso wie die Strukturierung der Software und die Aufteilung in Module. Das Softwaredesign sollte neben der Effizienz der Implementierung auch die in Abschnitt 1.1 genannten drei Qualitätspunkte Zuverlässigkeit, Wartbarkeit und Erweiterbarkeit verfolgen.

$\texttt{case}_l\ ys'\ \texttt{Nil}\ (\texttt{f}\ xs)$	$\doteq$	$\texttt{case}_l\ ys'\ \texttt{Nil}\ (\texttt{g}\ xs),\ (xs, ys')$
$\texttt{case}_l\ \texttt{bot}\ \ldots$	$\doteq$	$\texttt{case}_l\ \texttt{bot}\ \ldots,\ (xs, \texttt{bot})$
$\texttt{bot}$	$\doteq$	$\texttt{bot},\ (xs, \texttt{bot})$
$\texttt{case}_l\ \texttt{Nil}\ \texttt{Nil}\ \ldots$	$\doteq$	$\texttt{case}_l\ \texttt{Nil}\ \texttt{Nil}\ \ldots,\ (xs, \texttt{Nil})$
$\texttt{Nil}$	$\doteq$	$\texttt{Nil},\ (xs, \texttt{Nil})$
$\texttt{case}_l\ (\texttt{Cons}\ y\ ys)\ \texttt{Nil}\ (\texttt{f}\ xs)$	$\doteq$	
		$\texttt{case}_l\ (\texttt{Cons}\ y\ ys)\ \texttt{Nil}\ (\texttt{g}\ xs),\ (xs, \texttt{Cons}\ y\ ys)$
$\texttt{f}\ xs\ y\ ys$	$\doteq$	$\texttt{g}\ xs\ y\ ys,\ (xs, \texttt{Cons}\ y\ ys)$
$\texttt{Cons}\ y\ (\texttt{append}\ xs\ ys)$	$\doteq$	$\texttt{Cons}\ y\ xs,\ (xs, \texttt{Cons}\ y\ ys)$
$y \doteq y,\ \texttt{append}\ xs\ ys$	$\doteq$	$xs,\ (xs, \texttt{Cons}\ y\ ys)$
$\texttt{append}\ xs\ ys$	$\doteq$	$xs,\ (xs, \texttt{Cons}\ y\ ys)$
$\texttt{case}_l\ xs\ ys\ (\texttt{consappend}\ ys)$	$\doteq$	$xs,\ (xs, \texttt{Cons}\ y\ ys)$
$\texttt{case}_l\ \texttt{bot}\ \ldots$	$\doteq$	$\texttt{bot},\ (\texttt{bot}, \texttt{Cons}\ y\ ys)$
$\texttt{bot}$	$\doteq$	$\texttt{bot},\ (\texttt{bot}, \texttt{Cons}\ y\ ys)$
$\texttt{case}_l\ \texttt{Nil}\ ys\ \ldots$	$\doteq$	$\texttt{Nil},\ (\texttt{Nil}, \texttt{Cons}\ y\ ys)$
ys	$\doteq$	$\texttt{Nil},\ (\texttt{Nil}, \texttt{Cons}\ y\ ys)$
$\texttt{Nil}$	$\doteq$	$\texttt{Nil},\ (\texttt{Nil}, \texttt{Cons}\ y\ \texttt{Nil})$
$\texttt{append}\ (\texttt{Cons}\ z\ zs)\ ys$	$\doteq$	$\texttt{Cons}\ z\ zs,\ (\texttt{Cons}\ z\ zs, \texttt{Cons}\ y\ ys)$
$\vdots$	$\doteq$	$\vdots$
$z \doteq z,\ \texttt{append}\ zs\ ys$	$\doteq$	$zs,\ (\texttt{Cons}\ z\ zs, \texttt{Cons}\ y\ ys)$
$\texttt{append}\ zs\ ys$	$\doteq$	$zs,\ (\texttt{Cons}\ z\ zs, \texttt{Cons}\ y\ ys)$
$(\texttt{Cons}\ z\ zs, \texttt{Cons}\ y\ ys)\ \texttt{where}\ (zs, \texttt{Cons}\ y\ ys)\ =\ \mathit{AppBsp}$		

Abbildung 5.1: Ein Beispiel für die Gleichheitsanalyse

5.4.1 Ziele des Designs

Daher standen als wesentlichste Ziele folgende im Vordergrund:

1. Die Gleichheitsanalyse stellt ja im Prinzip eine Erweiterung der Kontextanalyse um Ausdrücke mit dem speziellen Prädikat `EQ` dar. Das Programm soll aber auch an andere Prädikate leicht angepaßt werden können.

2. Der Kontext- bzw. Gleichheitskalkül ist nichtdeterministisch. Er ist allein durch die Expansions- und Meta-Regeln spezifiziert, ohne daß in dem formalen System die Anwendung der Regeln in einer bestimmten Reihenfolge vorgeschrieben wird. Aus Gründen der Effizienz soll das Programm den Nichtdeterminismus nicht einfach simulieren, d.h. für eine Wurzel alle möglichen Gleichheitstableaus erzeugen und durchsuchen.

3. Die Gleichheitsanalyse ist prinzipiell auf jeder funktionalen Kernsprache mit algebraischen Datentypen und einer nicht-strikten Semantik durchführbar. Die verwendete Kernsprache soll daher auf einfache Weise gegen eine andere ausgetauscht werden können.

5.4.2 Anforderungen

Aus den genannten Zielen ergeben sich unterschiedliche Anforderungen:

zu 1) Um das Programm auch an andere Prädikate anpassen zu können, müssen die für die Gleichheitsanalyse spezifischen Teile strikt von denen der Kontextanalyse getrennt werden.

zu 2) Hier gilt es, eine Heuristik zu entwickeln, die entscheidet, wann welche Regel angewendet bzw. weiter verfolgt werden soll. Es ist aber nicht wünschenswert, diese Heuristik mit den Regeln zu vermengen, da sonst aus dem Programm nicht mehr ohne weiteres ersichtlich wäre, welche Prämissen für die Kalkülregeln gelten und welche Voraussetzungen für die Heuristik geschaffen wurden. Damit ginge die Übersichtlichkeit und der klare Bezug zur Theorie verloren.

zu 3) Die Tableauregeln dürfen nicht direkt von dem Datentypen, der die Kernsprache implementiert, abhängen. Es muß also von der konkreten Implementierung der Kernsprache abstrahiert werden. Dies wiederum bedeutet, daß wir allgemeine Operationen definieren müssen, die auf jeder Repräsentation der Kernsprache realisierbar sind.

5.4.3 Mögliche Realisierung

Eine bedeutendes Ziel hatten wir bisher noch nicht formuliert, obwohl es im Hinblick auf die Zuverlässigkeit selbstverständlich ist: Das Programm soll immer terminieren, auch wenn das Tableau unendlich groß werden kann, weil der Kalkül es nicht schließen kann, d.h. keine Lösung findet. Daher ist es erforderlich, daß ein Tableau stets nur endlich tief ausgewertet werden darf.

Da wir mit Haskell in einer verzögert auswertenden Sprache programmieren, müssen wir dies nicht unbedingt schon bei der Generierung des Tableaus berücksichtigen. Vielmehr können wir uns darauf beschränken, beim Zusammenstellen des Lösungskontextes eine maximale Tableautiefe nicht zu überschreiten.

Gliederung in Phasen

Um neben der o.g. auch die Anforderungen 1 und 2 zu realisieren, liegt nun die Aufteilung in folgende Schritte nahe:

1. Generierung des Tableaus mittels der Kalkülregeln
2. Heuristische Auswahl der vielversprechenden Regeln
3. Zusammenstellen der Lösung, endliche Auswertung

Dabei kann im 1. Schritt noch zwischen den Expansions- und den Meta-Regeln unterschieden werden, was die in Abbildung 5.2 zu erkennenden Phasen ergibt. Aus der gleichen Abbildung ist außerdem zu ersehen, daß

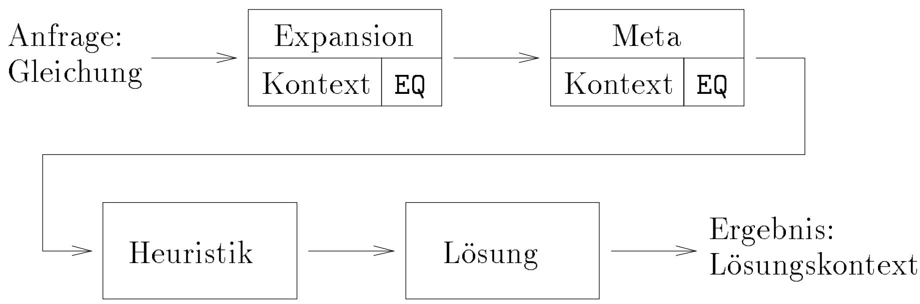

Abbildung 5.2: Die Phasen der Gleichheitsanalyse

die Tableauregeln am besten in verschiedene Gruppen eingeteilt werden. Die Benennung erfolgt entsprechend der Tabelle 5.1.

	Kontext	EQ
Expansionsregeln	ContextLocal	EQLocal
Meta-Regeln	ContextMeta	EQMeta

Tabelle 5.1: Einteilung der Tableauregeln

Datenstrukturen

Der 1. Schritt der Gleichheitsanalyse soll ja darin bestehen, aus einer Anfrage, die den Wurzelknoten darstellen wird, mit Hilfe der Expansionsregeln das Tableau aufzubauen. Da die heuristische Auswahl erst später erfolgt, stellt das Ergebnis der Expansion sogar Mengen von Tableaus dar. Die darauf folgende Anwendung der Meta-Regeln ist eine Operation auf Mengen von Tableaus. Erst durch die Heuristik wird aus einer solchen Menge von Tableaus ein bestimmtes ausgewählt, welches zwar noch potentiell unendlich sein kann, aber beim Zusammenstellen der Lösung nur endlich tief ausgewertet wird. Der Zusammenhang zwischen diesen Datenstrukturen und den

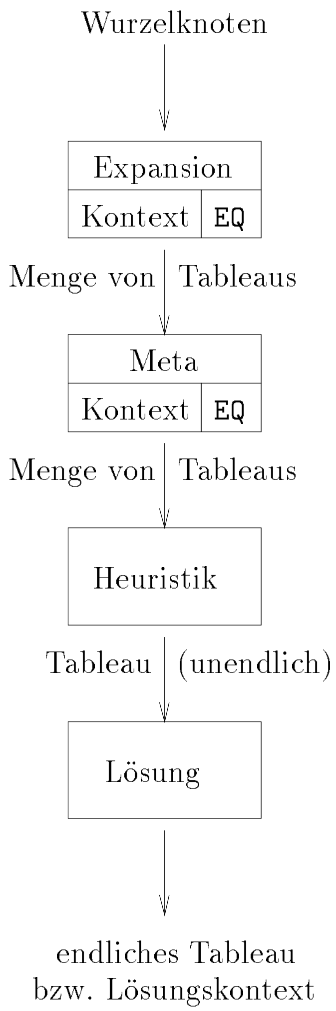

Abbildung 5.3: Die Datenstrukturen zwischen den einzelnen Phasen

einzelnen Phasen ist in Abbildung 5.3 zu sehen. Vor dem Hintergrund, daß jede Phase durch eine eigene Funktion implementiert werden soll, erhalten wir auch schon Hinweise auf die Typen dieser Funktionen. Außerdem kann man die *Pipeline* von Funktionsanwendungen sehr gut erkennen.

Die Kernsprache

Eine Anforderung wurde bisher noch nicht behandelt. Diese betrifft die Unabhängigkeit der Tableauregeln von der Kernsprache, die mit einem abstrakten Datentypen gelöst werden kann. Noch besser erscheint sogar eine eigene Typklasse `CoreExpression`, da dann verschiedene Repräsentationen eines Kernsprachenausdruckes gleichzeitig in einem Programm verwendet werden können. Wir müssen uns ja ohnehin überlegen, wie die Menge der für eine Kernsprache typischen Operationen geschickt zu definieren ist. Zu diesen Operationen sollten auf jeden Fall Funktionen zählen, die

- einen Reduktionsschritt in Normalordnung durchführen.
- die Menge $\mathcal{FV}(t)$ der freien Variablen eines Ausdruckes t berechnen.
- bestimmen, ob ein Ausdruck ein Redex ist, oder welcher Art eine WHNF ist, z.B. FWHNF oder CWHNF.
- Informationen über die Struktur eines Ausdruckes liefern, also z.B. den Konstruktor einer CWHNF oder den möglichen Termkontext, in dem sich ein `case` befindet.

5.5 Die Implementierung

Die Implementierung wird in die soeben vorgestellten Phasen gegliedert. Vorüberlegungen in Haskell-Syntax, die nicht im endgültigen Code auftauchen, fehlt das vorangestellte `>`, vgl. dazu Abschnitt 3.2. Module mit Hilfsfunktionen, die zwar von Interesse sind, deren Erörterung aber an dieser Stelle zu weit führen würde, sind, im Anhang B aufgeführt. Genauso wird an einigen Stellen auf die Darstellung von Details verzichtet, die zum Verständnis nicht wesentlich beitragen. Da nur die entsprechenden Teile nicht in den Text dieser Arbeit aufgenommen werden, bleibt das Programm davon unberührt und die vollständige Arbeit ist im World Wide Web über die Adresse `http://www.ki.informatik.uni-frankfurt.de/` zu finden.

5.5.1 Generierung des Tableaus

Bevor wir uns überlegen, welche Datenstruktur wir für eine *Menge* von Tableaus wählen, müssen wir erst einmal klären, wie man ein *einzelnes* Kontexttableau darstellen kann.

Darstellung eines Kontexttableaus

Dies ist im Grunde ein Baum mit Listen von Nachfolgern, die logisch eine *Disjunktion* bedeuten. Die Knoten enthalten eine Liste möglicher Bedingungen, hier mit `Constraints` bezeichnet, die ihrerseits als logische *Konjunktion* betrachtet werden, und eine Substitution. Dafür könnte man folgende Datenstruktur angeben:

```
data Tableau = Branch Label [Tableau]
             | Leaf Label ..
```

wobei

```
type Label = ([Constraint], Substitution)
```

Wie eben schon angedeutet, kann ein Kontexttableau als Und-Oder-Baum aufgefaßt und entsprechend implementiert werden. Man kann diese Und-Oder-Struktur aber auch auf eine ganz andere Weise hervorheben. Denn entweder hat ein Knoten einen Nachfolgers oder ist eben ein Blatt:

```
data Tableau = Root Label Successor
             | Leaf Label ..

type Successor = Various Tableau
```

mit einem Typen `Various`, der es erlaubt, gar keinen, genau einen oder mehrere alternative bzw. gleichzeitig gültige Werte zu repräsentieren.

```
data Various a = VNothing
               | VJust a
               | VOr [a]
               | VAnd [a]
```

Welche Vorteile hat die Darstellung nun? Wie bereits erwähnt, drückt sie die Und-Oder-Struktur explizit aus, was für die Implementierung der Regeln hilfreich sein sollte. Denn die Regeln weisen die gleiche Struktur auf: Manche können in einem bestimmten Fall keinen gültigen Wert zurückgeben, andere genau einen oder mehrere evtl. gleichzeitig gültige bzw. alternative Werte.

Darstellung von Mengen von Kontexttableaus

Was ist nun eine Menge von Tableaus? Üblicherweise benutzen wir für eine Menge gerne die Datenstruktur Liste, aber

```
type TableauSet = [Tableau]
```

gestaltet schon den Aufbau des Tableaus mit Hilfe der Expansionsregeln schwierig und aufwendig. Denn es müßte jedesmal ein Tableau der Liste entnommen und durch eine Menge von Tableaus ersetzt werden, die dann bis zu einer Tiefe n identisch sind. Aus diesem Grund ist die Datenstruktur auch für die Selektion einer Teilmenge von Tableaus denkbar schlecht.

Datenstruktur für Mengen von Tableaus Daher gilt es, in diesem Modul eine andere Datenstruktur für Mengen von Tableaus zu finden.

```
> module TableauSet(
>   TableauSet(..), takeTableauSet,
>   Various(..), listToVariousWith,
>   fromVarious, fromVarious1, isVNothing,
>   ExpansionRule(..),
>   expandTableau, loopTableau,
>   PathRule(..), PathObservation(..)) where
```

Es stellt sich als vorteilhaft heraus, die Menge von Tableaus als disjunkte Vereinigung von Mengen von Tableaus mit gleichen Wurzeln und unterschiedlichen Nachfolgern anzusehen, d.h.

$$T = \dot{\bigcup} T_w$$

wobei T_w die Menge von Tableaus mit Wurzel w ist. Wollen wir die T_w-Mengen unter einer einzigen Wurzel zusammenfassen, können wir die folgende Struktur benutzen, wobei wir den Datentypen sogleich polymorph über dem Knoteninhaltstyp halten:

```
> data TableauSet a = TSBranch [Various (TableauSet a)] a
>                   | TSLeaf a
>                   deriving (Show)
```

Wie bei vielen baumartigen Datenstrukten ist es auch für ein `TableauSet` nützlich, Instanz der `Functor`-Klasse zu werden und somit die Funktion `map` bereitzustellen.

```
> instance Functor TableauSet where
>   map f (TSBranch s a) = TSBranch (map (map (map f)) s) (f a)
>   map f (TSLeaf a)     = TSLeaf (f a)
```

Expansion des Tableaus. Die für `TableauSet` gegebene Definition von `map` greift auf ein `map` zurück, welches auf dem `Various`-Typen definiert sein muß:

```
> data Various a = VNothing
>                | VJust a
>                | VOr [a]
>                | VAnd [a]
>                deriving (Show)

> instance Functor Various where
>   map f (VNothing) = VNothing
>   map f (VJust x)  = VJust (f x)
>   map f (VOr xs)   = VOr (map f xs)
>   map f (VAnd xs)  = VAnd (map f xs)
```

Im Prinzip scheint der Konstruktor `VJust` überflüssig, da er durch eine einelementige Liste sowohl bei `VOr` als auch bei `VAnd` abgedeckt werden kann. Benutzt man aber immer die Funktion `listToVariousWith`, wenn man aus einer Liste ein Element vom Typ `Various` konstruieren möchte, dann wird ein einzelner Wert eindeutig durch `VJust` dargestellt, und ein Pattern Match darauf ist auf konsistente Weise möglich.

```
> listToVariousWith _    []    = VNothing
> listToVariousWith _   [x]    = VJust x
> listToVariousWith con xs     = con xs
```

Wird ein `Various`-Element mittels `listToVariousWith` konstruiert ist außerdem die folgende Hilfsfunktion `fromVarious1` immer definiert, vorausgesetzt, es handelt sich nicht um `VNothing`.

```
> fromVarious1 (VJust a)      = a

> fromVarious1 (VOr (a:as))   = a
> fromVarious1 (VOr ([]))     = error "...fromVarious1: VOr []"

> fromVarious1 (VAnd (a:as))  = a
> fromVarious1 (VAnd ([]))    = error "...fromVarious1: VAnd []"

> fromVarious1 VNothing       = error "...fromVarious1: VNothing"
```

Um gerade diesen letzten Fall abzufragen, können wir die Funktion `isVNothing` benutzen:

```
> isVNothing VNothing         = True
> isVNothing _                = False
```

Nun kann der Typ einer Expansionsregel schon einmal festgelegt werden. Wie schon erwähnt, handelt es sich um Funktionen, die zu einen Knoteninhalt verschiedene Inhalte zurückliefern:

```
> type ExpansionRule a = (a -> Various a)
```

Damit läßt sich die `expandTableau`-Funktion schließlich recht kompakt formulieren. Mit dieser generieren wir genaugenommen gar keine Menge von Tableaus, sondern *überführen* eine Menge in eine neue unter Zuhilfenahme von Regeln, die nur die *lokale* Information innerhalb eines Knotens benötigen. Im Regelfall wird man die Funktion `expandTableau` aber mit einem `TSBranch`, der eine leere Nachfolgerliste hat, aufrufen, wie es auch in der letzten Zeile selbst geschieht.

```
> expandTableau :: (a -> Bool) -> [ExpansionRule a] ->
>                  TableauSet a -> TableauSet a

> expandTableau isLeaf rules tableau@(TSLeaf _) =
>   tableau

> expandTableau isLeaf rules tableau@(TSBranch succ's root)
>   | isLeaf root = TSLeaf root
>   | otherwise   =
>     TSBranch (succ's ++ [ map f (rule root) | rule <- rules ])
>            root
>   where
>     f = expandTableau isLeaf rules . TSBranch []
```

Wie angedeutet, wird also die Nachfolgerliste `succ's` eines bestehenden Tableaus `tableau` einfach um weitere mögliche Nachfolger ergänzt. Diese ergeben sich durch die Anwendung jeder Regel aus `rules` auf den Inhalt `root` des Wurzelknotens.

Um aus den zurückgegebenen — verschiedenen — Konteninhalten wieder Tableaus zu erstellen und den gesamten Prozess der Expansion rekursiv fortzusetzen, wird die Funktion `expandTableau isLeaf rules . TSBranch`

`[]` in Verbindung mit `map`[12] benutzt. Dabei erweist es sich als Vorteil, gerade diese Reihenfolge der Argumente für den `TSBranch`-Konstruktor gewählt zu haben, da nun `(TSBranch []) :: a -> TableauSet a` ist, d.h. genauso wie `TSLeaf` eine Funktion, die einen Knoteninhalt nimmt und daraus ein `TableauSet` erzeugt.

Darüberhinaus erhält die `expandTableau`-Funktion als Argument eine Funktion `isLeaf`, die Auskunft darüber geben soll, ob ein vorliegender Knoteninhalt schon für ein Blatt genügt.

5.5.2 Schließen des Tableaus

Nach Definition 5.3.1 ist ein Tableau geschlossen, wenn alle Blätter gelöst sind. Nicht immer ist es möglich, die Bedingungen mittels der Regel (SyntEQ) aus den Knoten zu entfernen. Daher kann es notwendig sein, nach Schleifen zu suchen, um das Tableau zu schließen.

Schleifenerkennung

Es wurde bereits angesprochen, daß die Funktion `expandTableau` eine Operation aufgrund *lokaler* Informationen darstellt.

Die Regeln zur Schleifenerkennung operieren offensichtlicherweise nicht lokal. Denn um eine Schleife zu bilden, ist Wissen über die zwei Knoten, die durch die Schleife verbunden werden sollen, und den dazwischenliegenden Pfad nötig. Außerdem kann es vorkommen, daß Informationen über den Pfad von der Wurzel zum ersten Knoten der Schleife vorliegen müssen.

Damit ist eine Loop-Regel nicht mehr wie eine Expansionsregel eine Funktion von Knoten auf eventuell mehrere Knoten, sondern eine Funktion, die von zwei Knoten und deren Verbindungspfad sowie dem Pfad von der Wurzel bis zum ersten Knoten abhängt.

Nutzung der Informationen für verschiedene Regeln Wir gehen davon aus, daß wir durchaus mehrere verschiedene Regeln zur Bildung einer Schleife überprüfen müssen. Nun ist es aber keineswegs effizient, für jedes Knotenpaar und den dazwischenliegenden Pfad jede mögliche Loop-Regel einfach von Neuem auszuprobieren. Vielmehr sollten die Informationen, die für eine bestimmte Loop-Regel herausgefunden wurden, auch für die anderen wiederverwendet werden und nicht abermals berechnet werden.

Zum Beispiel müssen die beiden Knoten i.a. mit Hilfe einer Substitution ineinander übergeführt werden können. Diese Eigenschaft bezeichnen wir ja nach Definition 5.2.3 als einen Match. Haben wir einmal festgestellt, daß die betreffenden Knoten nicht matchen, so ist für diese kein Loop möglich. Auf der anderen Seite können wir uns im Falle eines Match die entsprechende Substitution merken und die einzelnen Pfadbedingungen prüfen. Im

[12] auf dem `Variuos`-Typen

Gegensatz zu der Forderung, die an die Knoten gestellt wird, differieren die Pfadbedingungen.

Nutzung von Informationen für verschieden lange Pfade Generell kann man unterscheiden zwischen Bedingungen, die erst *nach* dem Auftreten gewisser Merkmale erfüllt sind, sowie solchen, die erfüllt sind, *solange* ein Merkmal *nicht* auftritt.

Auch hier ist es sinnvoll, sich Gedanken darüber zu machen, wie Mehrfachberechnungen vermieden werden können. Dies soll am Beispiel eines Pfades $e_1 = (v_0, v_1), e_2 = (v_1, v_2), e_3 = (v_2, v_3), e_4 = (v_3, v_4)$ verdeutlicht werden[13]. In diesem Pfad soll mindestens links und rechts eine Reduktion vorkommen, in Zeichen L bzw. R. Angenommen e_2 habe Eigenschaft R und e_4 Eigenschaft L. Dann müßte für eine Funktion `pathRule` folgendes gelten:

- `pathRule` [e_1] `= False`
- `pathRule` [e_1, e_2] `= False`
- `pathRule` [e_1, e_2, e_3] `= False`
- `pathRule` [e_1, e_2, e_3, e_4] `= True`
- `pathRule` [$e_1, e_2, e_3, e_4, \ldots$] `= True`

Nachdem R und L aufgetreten sind, bleibt der Funktionswert immer `True`. Schwer ist es nicht, dies zu realisieren, wenn die Funktion `pathRule` stets die gesamte Liste durchläuft. Aber es ist auch leicht zu erkennen, daß wir dabei sehr verschwenderisch mit Rechenzeit umgehen würden. Es wäre zwar elegant, wenn die Funktion immer nur eine Kante nähme, und ein Paar lieferte, bestehend aus

- dem für den bisherigen Teilpfad zutreffenden Wahrheitswert und
- einer Funktion, die man auf die folgende Kante des Pfades angewenden kann.

Aber eine solche Funktion, die ein Paar aus einem Wahrheitswert und einer Funktion zurückgibt, die ein Paar aus einem Wahrheitswert und einer Funktion zurückgibt, die ..., läßt sich statisch nicht typen, weil es sich dabei — wie man sieht — um einen unendlichen Typ handelte.

Hingegen können wir den Gedanken aufgreifen, daß immer nur eine weitere Kante konsumiert werden soll. Denn das Vorgehen bei der Anwendung der Meta-Regeln sieht ja allgemein so aus, daß wir vom Wurzelknoten ausgehend *von oben* in den Baum, d.h. das `TableauSet`, hinabsteigen. Dieses Absteigen führt nun ohnehin alle Pfade entlang, so daß wir dort auch Buch darüber führen sollten, welche Eigenschaften für eine bestimmte Loop-Regel erfüllt bzw. verletzt werden.

[13] Die e_i sind die Kanten und die v_i die Knoten.

Repräsentation der Pfadregeln Anstatt direkt mit Funktionen zu arbeiten, müssen wir die Pfadregeln eben durch einen algebraischen Typen darstellen. Hierbei ist aber nicht so sehr der Typ selbst interessant, so daß wir die Details seiner Implementierung fortlassen, sondern vielmehr die Operation, die auf ihm definiert sein muß. Diese Operation — wir nennen sie `observePath` — bekommt als Argumente einen Knoten sowie eine Regel von diesem Typ und gibt ein Paar aus

- dem für den bisherigen Teilpfad zutreffenden Wahrheitswert und
- einer Regel, die für die folgenden Knoten eines Pfades angewendet werden kann.

zurück. Da man diese Spezifikation sicherlich mit unterschiedlichen Typen realisieren kann, wählen wir die Definition über eine Typklasse.

```
> class PathRule rule where
>   observePath :: a -> rule a -> (Bool, rule a)
```

Funktionen für die Schleifensuche Nun können wir uns der Frage zuwenden, wie am einfachsten alle möglichen Schleifen gefunden werden können. Wenn später die Heuristik auf das `TableauSet` zugreifen soll, wird dies wiederum über dessen Wurzel r erfolgen. Dabei ist es dann bedeutend, ob ein möglicher Nachfolger s schon ein Blatt ist, weil er bereits eine Schleife mit der Wurzel bildet. Denn so kommen Schleifen unterhalb dieses Knotens s gar nicht mehr in Betracht.

Das heißt, die Suche nach Schleifen kann ohne weiteres so programmiert werden, daß für eine Wurzel zuerst Schleifen mit allen möglichen direkten Nachfolgern der nächst tieferen Ebene gesucht werden, dann mit Knoten in der übernächsten Ebene usf. In den Tableaus, bzw. Mengen von Tableaus, in die dabei hinabgestiegen wird, sind die möglichen Schleifen, die ihren Ausgangspunkt wiederum in tieferen Ebenen haben, schon gebildet worden.

```
> loopTableau :: PathRule rule => (a -> a -> Maybe a) ->
>                [rule a] -> [rule a] ->
>                TableauSet a -> TableauSet a

> loopTableau match rootRules rules tableau@(TSLeaf _) =
>   tableau

> loopTableau match rootRules rules tableau@(TSBranch succ's root)
>    | or preds  = TSBranch (map (map makeNewSucc) succ's) root
>    | otherwise = tableau
```

```
>   where
>     (preds, _)    =
>       unzip [ (observePath root rule) | rule <- rootRules ]
>     makeNewSucc t = loopsWith match rules root []
>                       (loopTableau match rootRules rules t)
```

Die Funktion `loopTableau` erledigt die Aufgabe, alle Schleifen für die Menge von Tableaus zu finden, die die Wurzel `root` haben. In der letzten Zeile findet man den rekursiven Aufruf auf den Nachfolgern. Dieser sorgt dafür, daß dort bereits Schleifen gebildet werden, bevor die Funktion `loopsWith` aufgerufen wird, um die möglichen Schleifen vom Wurzelknoten in die verschieden tiefen Ebenen hinein zu suchen.

```
> loopsWith match rules node path tableau@(TSLeaf _) =
>   tableau

> loopsWith match rules node path tableau@(TSBranch succ's root)
>   | or preds  = maybe newTableau TSLeaf (match node root)
>   | otherwise = newTableau
>   where
>     newTableau = TSBranch (map (map makeNewSucc) succ's)
>                           root
>     (preds, funcs) =
>       unzip [ (observePath root rule) | rule <- rules ]
>     makeNewSucc = loopsWith match funcs node (root:path)
```

Diese Vorgehensweise, die in Abbildung 5.4 noch einmal grafisch veranschaulicht wird, berechnet nicht mehr Schleifen als notwendig, weil das Tableau ohnehin nur so weit ausgewertet wird, wie es benötigt wird.

5.5.3 Die analysierte Kernsprache

Wie bereits in Abschnitt 5.4.3 erwähnt, wird die konkrete Repräsentation der Kernsprache in der Implementierung durch eine Typklasse abstrahiert. Dies geschieht im Modul `CoreLanguage`:

```
> module CoreLanguage(
>   module CoreLanguage, compile) where
```

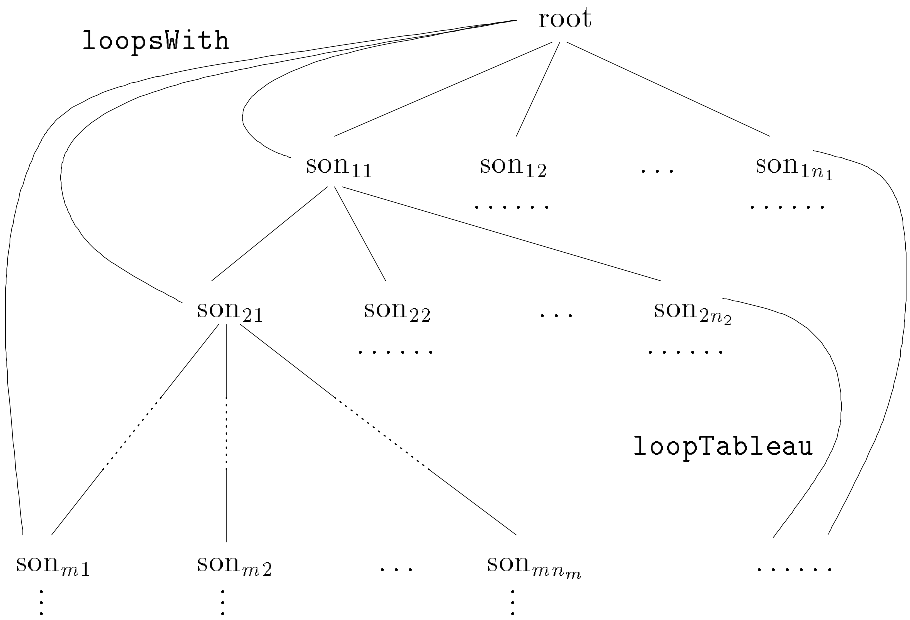

Abbildung 5.4: Das Zusammenspiel der Funktionen `loopTableau` und `loopsWith` bei der Schleifensuche

Die Typklasse `CoreExpression`

Macht man sich Gedanken über den Aufbau dieser Klasse, so wird schnell deutlich, daß für die meisten möglichen Instanzen eine *Umgebung* benötigt wird, in der z.B. die Informationen über die Definitionen von Superkombinatoren gespeichert werden müssen. Möchte man z.B. wissen, ob ein Ausdruck e eine FWHNF ist, so ist folgende Vorgehensweise denkbar: Falls es sich um eine Applikation handelt, muß der Umgebung die Information entnommen werden, ob es sich bei dem Operator um einen Superkombinator handelt, und wenn ja, wie groß dessen Stelligkeit ist.

Als erstes sollen im folgenden Funktionen angegeben werden, die Informationen über die Art des vorliegenden Ausdruckes liefern. Wie man sieht, spiegelt sich schon bei diesen Definitionen die Notwendigkeit einer Umgebung wider.

```
> class (Eq e) => CoreExpression e where
>   isWHNF    :: e -> Env -> Bool
>   isFWHNF   :: e -> Env -> Bool
>   isCWHNF   :: e -> Env -> Bool
>   isSCWHNF  :: e -> Env -> Bool
>   isPREDEX  :: e -> Env -> Bool
```

```
>   isREDEX   :: e -> Env -> Bool

>   isFreeVar :: e -> Env -> Bool
```

Dabei soll `isPREDEX` — abweichend zu Definition 5.2.1 – auf genau den Ausdrücken wahr sein, die ein `case` auf eine freie Variable darstellen[14]. Die Bedeutung der übrigen Funktionen sollte unmittelbar einleuchten. Die konkrete Realisierung des Typs `Env`, der für die Umgebung steht, wird später besprochen.

Die nächsten Funktionen sind solche, die aus einem Ausdruck bestimmte Komponenten isolieren können. Wir brauchen sie, um z.B. auf den Namen einer freien Variablen zugreifen zu können oder auf Bezeichner[15], Stelligkeit und Argumente eines Konstruktors bzw. Name, Stelligkeit und Argumente einer Superkombinatoranwendung:

```
>   getFreeVar       :: e -> Env -> Maybe Name
>   getConstructor   :: e -> Env -> Maybe (Tag,  Arity, [e])
>   getSupercomb     :: e -> Env -> Maybe (Name, Arity, [e])
>   getCase          ::
>     e -> Env ->
>     [(e, [(Tag, Arity, [e] -> StateTransformer Env e)])]
```

`getCase` liefert zu einem Ausdruck e und seiner Umgebung die möglichen Programmkontexte, in denen ein `case` auftaucht. Die zweite Komponente in der Liste von Paaren stellt dabei die Alternativen dar, soll aber nicht im Detail erörtert werden.

Zu den in Abschnitt 5.4.3 genannten Operationen kommen nun noch solche hinzu, mit denen *neue* Ausdrücke konstruiert werden können.

```
>   mkFreeVar     :: Name   -> StateTransformer Env e
>   mkApplication :: e -> e -> StateTransformer Env e
>   mkConstructor ::
>     (Tag,  Arity, [e]) ->
>     StateTransformer Env e
>   mkCase        ::
>     (e, [(Tag, Arity, [e] -> e)]) ->
>     StateTransformer Env e
```

[14] Eine abstrakte Applikationen nach Definition 5.2.2 stellt zwar auch einen potentiellen Redex dar, wird aber gesondert behandelt

[15] Im folgenden wird auch häufig das englische Wort *tag* dafür benutzt.

An dieser Stelle muß der Typ `StateTransformer` erklärt werden, der schon bei `getCase` auftauchte. Er kapselt die Tatsache, daß sich die Umgebung verändern kann, wenn wir einen neuen Ausdruck bilden. Wenn z.B. die Funktion `mkApplication` auf zwei Ausdrücke angewendet werden soll, stellt der zurückgegebene `StateTransformer Env e` im Prinzip die Funktion dar, die in einer Umgebung den Ausdruck konstruiert und diesen mitsamt der neuen Umgebung zurückliefert. Der Typ `StateTransformer` ist in einem eigenen Modul definiert, wo er auch zu einer Instanz der Klasse `Monad` gemacht wird, was uns später die Notation bei der sukzessiven Konstruktion von Ausdrücken erleichtern wird. Dieses Modul `StateMonad` ist im Anhang B.3 zu finden.

Benutzt man zur Implementierung der Klasse `CoreExpression` eine Struktur, die Sharing gewährleistet, so verändert ein Reduktionsschritt zwangsläufig die jeweilige Umgebung:

```
>   reduce      :: e -> StateTransformer Env e
>   reduceN     :: e -> Env -> [(e, Env)]
```

Dabei stellt `reduce` einen Reduktionsschritt in Normalordnung dar und `reduceN` kann als eine nichtdeterministische Version, die mehrere mögliche Ergebnisse liefern kann, angesehen werden.

Nun folgen noch einige Funktionen, die auf Ausdrücken operieren und in den Kalkülregeln benötigt werden, wie z.B. die Anwendung einer Substitution oder eine Funktion, die die Liste aller freien Variablen bzw. die Liste der Namen aller freien Variablen eines Ausdruckes liefert.

```
>   applySubst ::
>     Substitution e e -> e ->
>     StateTransformer Env e

>   fv          :: e -> Env -> [e]
>   fvNames     :: e -> Env -> [Name]

>   match       ::
>     (e, Env) -> (e, Env) ->
>     Maybe (Substitution Name e)

>   syntEq      :: e -> e -> Env -> Bool
```

Die Funktion `match` dient dazu, festzustellen, ob zwei Ausdrücke, die in ihren jeweiligen Umgebungen interpretiert werden, mittels einer Substitution auf den freien Variablen ineinander überführt werden können.

Außerdem wird natürlich eine Funktion benötigt, die Auskunft darüber geben kann, ob zwei Ausdrücke syntaktisch gleich sind. Diese kann man direkt mittels `match` angeben, wie man in der Definition der Default-Methoden sieht:

```
>   syntEq ex1 ex2 env =
>     maybe False
>       (== emptySubst)
>       (match (ex1, env) (ex2, env))

>   isWHNF   e = (isFWHNF e) .||. (isCWHNF e)
>   isFWHNF  e = (isWHNF  e) .&&. (not . isSCWHNF e)
>   isSCWHNF e = (isCWHNF e) .&&. (not . isFWHNF e)

>   getFreeVar expr env
>     | isFreeVar expr env = Just (head (fvNames expr env))
>     | otherwise          = Nothing

>   reduceN expr env =
>     [ (reduce expr) `beginningWith` env ]

>   fvNames expr env =
>     mapMaybe ((flip getFreeVar) env) (fv expr env)
```

`.||.` und `.&&.` sind dabei Kombinatoren, die nicht zwei Boolesche Werte sondern zwei einstellige Funktionen, die jeweils einen Booleschen Wert zurückgeben, verknüpfen. Damit ist es möglich die Funktionen `isWHNF`, ... recht elegant mit nur einem Argument zu schreiben, obwohl sie noch nach einem zweiten, nämlich einer Umgebung, verlangen. Die Funktionen `.||.` und `.&&.` sind neben anderen nützlichen Kombinatoren in Anhang B.1 aufgeführt.

Die Umgebung

Nun kommen wir zurück auf den Typ `Env`, der die Umgebung repräsentiert. Dieser ist über ein Typsynonym definiert, d.h. er ist nur ein anderer Name für eine Datenstruktur, die für die in dieser Arbeit gewählten Instanzen von `CoreExpression` als Umgebung geeignet ist.

Das ist sicherlich *nicht* die eleganteste Lösung, da für andere Instanzen von `CoreExpression` die Definition des Typen `Env` geändert werden muß. Außerdem kann die Flexibilität mehrerer verschiedener Instanzen für `CoreExpression` in einem Programm nur noch durch recht umständliche

Konstruktionen gewährleistet bleiben[16].

Um die trotz dieser Nachteile für ein Typsynonym getroffene Entscheidung zu begründen, sollen im folgenden kurz die Alternativen beleuchtet werden.

Umgebung im Ausdruck Zum einen bestünde die Möglichkeit, daß jeder Ausdruck seine gesamte Umgebung selbst beinhaltet. Wenn dann aber zwei Ausdrücke jeweils eine Umgebung beinhalten, ist nicht klar, welche Umgebung z.B. für die Applikation dieser beiden Ausdrücke gelten sollte. Auch eine Regel, die einer bestimmten Umgebung, etwa der des ersten Ausdrucks, dabei implizit den Vorrang einräumt, scheint nicht befriedigend, weil sie vielleicht nicht immer offensichtlich ist. Das bedeutet, daß es dem Anwender der Funktion `mkApplication` nicht bewußt sein mag, daß u.U. Informationen über Superkombinatoren aus der Umgebung des zweiten Ausdruckes verlorengehen.

Die gewählte Darstelllung hat demgegenüber den Vorteil, daß die Umgebung stets *explizit* genannt wird. Daher muß der Programmierer sich vorher Gedanken darüber machen, welche Umgebung er angibt, und eventuell erst eine neue konstruieren, um aus den beiden vorhandenen alle gewünschten Informationen bereitzuhalten.

Schaut man sich `mkConstructor` an, so wird dies noch um einiges deutlicher. Denn wie könnte man mit dieser Funktion, wenn sie nicht ausdrücklich mit einer Umgebung versorgt würde, einen Ausdruck erzeugen, der einen 0-stelligen Konstruktor darstellt? Da in diesem Fall die Liste der Argumente leer bleibt, hätten wir keine Umgebung zur Verfügung, auf die wir zugreifen könnten.

Hieran ist zu erkennen, warum diese Möglichkeit nach reiflicher Überlegung ausgeschieden ist. Die nun folgenden Schilderungen setzen einige Kenntnisse über Typklassen von Haskell voraus, die in [PHA+97, S.31ff.] nachzulesen sind.

Freie Typvariable Ein weiterer Gedanke bestand aus dem Versuch, die Umgebung relativ frei zu belassen, indem dafür eine Typvariable angegeben wird:

```
class (Eq expr) => CoreExpression expr where
  isWHNF :: expr -> env -> Bool
```

Dies unterscheidet sich von der vorigen Definition erst einmal nur dadurch, daß `env` als Typvariable jetzt kleingeschrieben wird.

[16] Z.B. dadurch daß man *alle möglichen* Typen von Umgebungen in `Env` hineinpackt, und die passende für die jeweilige `CoreExpression`-Instanz herausgesucht wird.

Es ist nun aber nicht möglich, eine Instanz vom Typ `ExprInstance` für die Klasse `CoreExpression` anzugeben, wenn man dabei *einen bestimmten* Typ, nennen wir ihn `EnvInstance`, für die Umgebung benutzen möchte. Denn die Funktion `isWHNF` z.B. wird in der Instanzendefinition den allgemeinsten Typ `isWHNF :: ExprInstance -> env -> Bool` haben. Will man dann `isWHNF :: ExprInstance -> EnvInstance -> Bool` definieren, so stellt dies eine unzulässige Einschränkung dar.

Mehrparameterklassen Es gäbe eine Möglichkeit, diesen Ansatz zu realisieren, wenn man in der Klassendeklaration spezifizieren könnte, daß `env` immer *zusammen* mit einer speziellen Instanz für `expr` gewählt wird. Mit Typklassen, die mehrere Parameter erlaubten, wäre dies machbar.

```
class (Eq expr) => CoreExpression expr env where
  isWHNF :: expr -> env -> Bool
  ...

instance CoreExpression ExprInstance EnvInstance where
  ...
```

Es wäre eine konsistente Lösung und würde den elegantesten Ansatz darstellen. Leider sind Mehrparameterklassen in Haskell nicht erlaubt, auch wenn es schon Überlegungen gibt, sie später einmal zu integrieren, siehe dazu u.a. [JJM97].

Andere Möglichkeiten Daneben gab es auch Überlegungen, ob und wie man die Wirkung einer Mehrparameterklasse nachbilden könnte. Die Mittel, mit denen man das tun könnte, geraten aber schnell recht komplex. Dazu sei hier exemplarisch die Möglichkeit angeführt, für Umgebungen eine Typklasse `Environment` einzurichten mit Funktionen, die den Zugriff darauf nach bestimmten Kriterien erlauben.

```
class Environment env where
  ...

class (Eq expr) => CoreExpression expr where
  isWHNF :: (Environment env) => expr -> env -> Bool
```

Nun muß das Zusammenspiel zwischen den beiden Klassen `CoreExpression` und `Environment` erklärt werden. In der Regel wird es so aussehen, daß `Environment` u.a. eine Funktion bereitstellt, mit der man in der Umgebung nach der Definition eines Superkombinators suchen lassen kann. Dies kann z.B. über den Namen des Superkombinators geschehen. Offensichtlich ist, daß der Name dann auch in der gleichen Weise in dem Ausdruck gespeichert sein muß. Damit ist aber sofort eine Abhängigkeit der beiden Typen, die wir zu Instanzen der jeweiligen Klassen machen möchten, verbunden. Diese muß im Typsystem von Haskell explizit ausgedrückt werden, da sonst wiederum unzulässig eingeschränkte Typen auftauchen.

```
class Environment env where
  ... (env a) ...

class (Eq expr) => CoreExpression expr where
  isWHNF :: (Environment env) => (expr a) -> (env a) -> Bool
```

Da es schwer abzusehen ist, welche weiteren Zusammenhänge noch zu Tage treten, scheidet auch diese Alternative aus. Im Rahmen dieser Diplomarbeit wäre der Aufwand, der für die allgemeine Verwendbarkeit der Lösung getrieben werden müßte, einfach zu hoch. Dies erscheint umso einleuchtender vor dem Hintergrund, als daß der mit dem Typsynonym beschrittene Weg recht einfach auf eine Mehrparameterklasse umzusetzen ist, wenn diese erst einmal in Haskell verfügbar sind.

Die benutzte Instanz für Kernsprachenausdrücke

Wie die konkrete Instanz für die Klasse `CoreExpression` in dieser Arbeit aussieht, wurde bisher ausgeklammert. Für die Implementierung der beschriebenen Methoden ist prinzipiell auch die Datenstruktur des Syntaxbaumes denkbar, direkt oder in leicht abgewandelter Form. Wie der Name schon sagt, sind die Ausdrücke dann aber durch Bäume dargestellt, so daß das Sharing bei der Reduktion verlorengeht. Um eine effiziente Reduktion zu gewährleisten, möchten wir daher eine Implementierung, die die Graphstruktur der Ausdrücke respektiert. Dafür gibt es verschiedene Ansätze; eigentlich könnte jede abstrakte Maschine, die für die Compilierung verzögert auswertender funktionaler Sprachen entworfen wurde (vgl. [Buc95]), dazu dienen.

Die Template Instantiation Machine Unter diesen stellt die *Template Instatiation Machine*, zu deutsch *Muster Instanziierungsmaschine*, die einfachste Möglichkeit dar, siehe auch [PJL91]. Da sie im Rahmen der Terminierungsanalyse (vgl. [Kic96]) schon implementiert wurde, bot es sich an, sie auch für diese Arbeit zu verwenden. Ebendort und in [PJL91] wird sie ausführlich beschrieben, so daß auf eine wiederholende Darstellung verzichtet wird. Es seien nur kurz die wichtigsten Bestandteile eines Zustandes der Template Instanziierungsmaschine erläutert.

Heap: Hier werden die Ausdrücke in ihrer Graphenstruktur gespeichert.

Stack: Er repräsentiert den aktuellen Zustand der Auswertung, indem er das *Rückgrat*[17] des Ausdrucks aufnimmt.

Dump: Dieser dient zum Sichern von Stacks, wenn Zwischenauswertungen nötig werden.

Darüberhinaus gibt es noch eine Komponente, die globale Informationen über Superkombinatordefinitionen enthält, also ein Teil der Umgebung im oben diskutierten Sinne.

Die Terminierungsanalyse aus [Kic96] arbeitet genauso wie die Striktheitsanalyse in [Sch94] und die Gleichheitsanalyse dieser Arbeit mit Hilfe von abstrakter[18] Reduktion. Deren Bedeutung stimmt aber in diesen drei Bereichen nicht vollkommen überein. In der Striktheitsanalyse werden die abstrakten Werte `Top` und `Bot` eingeführt, in der Terminierungsanalyse sind es `Bot` und abstrakte Variable $\mathtt{T}^{x_i}$. Diese zeigen dort ein spezielles Verhalten, weshalb sie für die Gleichheitsanalyse nicht benutzt werden konnten. Stattdessen wurde die Template Instanziierungsmaschine dahingehend erweitert, daß sie Ausdrücke mit freien Variablen, die wie auch in [SS96a, S. 4] als abstrakte Ausdrücke bezeichnet werden[19], reduzieren kann. Die Rolle von `bot` hat sich gegenüber der Terminierungsanalyse nicht verändert, abgesehen von der Kleinschreibung, die deutlich machen soll, daß es sich hierbei um einen vordefinierten Superkombinator handeln soll und nicht um die abstrakte Darstellung einer Menge von Termen.

Die Implementierung der Klassenmethoden soll nicht in allen Details dargestellt werden, weil sie dann unverhältnismäßig viel Platz einnehmen würde. Es soll aber ein grober Überblick gegeben werden, wie die verschiedenen Funktionen realisiert wurden. Das bedeutet, daß von gleichartigen Funktionen jeweils eine examplarisch vorgestellt wird.

Einen Ausdruck und den aktuellen Zustand seiner Auswertung halten wir in folgender Datenstruktur fest:

```
> data TiExpr = TiExpr {
>   tiExprStack :: TiStack,
```

[17] engl. *spine*

[18] Leider kommt der Begriff „abstrakt“ in diesem Abschnitt in zwei unterschiedlichen Bedeutungen vor: zum einen im Sinne einer abstrakten, d.h. virtuellen Maschine, zum anderen im Bezug auf die abstrakte Reduktion, also Reduktion mit speziellen, abstrakten Werten.

[19] vgl. Abschnitt 4.3

```
>   tiExprDump  :: TiDump,
>   tiExprAddr  :: Addr
> } deriving (Eq, Show)
```

Die beiden ersten Komponenten entsprechen denjenigen im Zustand der Template Instanziierungsmaschine. Die letzte, `tiExprAddr`, stellt die Addresse des Ausdrucks im Heap dar. Sie ist redundant, weil man sie auch aus Stack und Dump ermitteln kann, aber der Zugriff darauf ist auf diese Weise wesentlich einfacher.

Die Grundlage, um die Informationen über die Art des Ausdruckes zu erlangen, bildet die Funktion `examineTiState`, die einen Zustand der Template Instanziierungsmaschine analysiert. Wir begnügen uns an dieser Stelle mit ihrem Typen:

```
> examineTiState :: TiState -> StateDesc
```

Die Beschreibung eines Zustandes erfolgt über die Struktur `StateDesc`,

```
> data StateDesc = StateDesc {
>   exprType :: ExprType,
>   node     :: Node,
>   spine    :: TiStack
> }
```

in der mehrere Informationen enthalten sind:

- `exprType` gibt Auskunft über die Art des Ausdruckes, also ob es ein Redex oder welche Art von WHNF es eventuell ist.

  ```
  > data ExprType = UNKNOWN | ABSAPP | FREEVAR | REDEX
  >               | PREDEX  | FWHNF  | CWHNF   | SCWHNF
  >                deriving (Show, Eq, Ord)
  ```

  Bis auf `UNKNOWN`, welches einfach dafür stehen soll, daß keiner der übrigen Typen ermittelt werden konnte, bleibt nur noch `ABSAPP` zu erklären. Hierbei handelt es sich um eine abstrakte Applikation, d.h. die Anwendung einer freien Variablen auf einen Term. Die übrigen Namen erklären sich von selbst, auch anhand der anfangs vorgestellten `is...`-Funktionen.

- `node` ist der Knoten aus dem Heap, von dem die vorige Entscheidung abhängig gemacht wurde.
- `spine` ist das Rückgrat des Ausdruckes, wie es abgerollt werden mußte, um an den genannten Heap-Knoten zu gelangen.

Die Untersuchung können wir auch auf einen Ausdruck beziehen, wenn wir ihn mit einem Zustand als Umgebung ausstatten:

```
> examineTiExpr :: TiExpr -> TiState -> StateDesc
> examineTiExpr tiExpr =
>   examineTiState . tiExpr2tiState tiExpr
```

`tiExpr2tiState` ist dabei eine Funktion, die den Ausdruck und seine Umgebung auf offensichtliche Weise kombiniert. Da wir nun die Implementierung der ersten Funktionen direkt angeben können, richten wir `TiExpr` gleich als Instanz von `CoreExpression` ein:

```
> tiExprType expr = exprType . examineTiExpr expr

> type Env = TiState

> instance CoreExpression TiExpr where
>   isSCWHNF expr  = (== SCWHNF) . tiExprType expr
>   isCWHNF expr   = (>= CWHNF)  . tiExprType expr
>   isFWHNF expr   = (== FWHNF)  . tiExprType expr
>   isWHNF expr    = (>= FWHNF)  . tiExprType expr

>   isPREDEX expr  = (== PREDEX) . tiExprType expr
>   isREDEX expr   = (== REDEX)  . tiExprType expr

>   isFreeVar expr = (== FREEVAR) . tiExprType expr
```

An sich ist die Eigenschaft „frei“ keine, die von einer Variablen selbst abhängt, sondern von dem Ausdruck, in dem sie eben frei oder gebunden auftaucht. In dieser Arbeit wird aber davon ausgegangen, daß diese Information schon bei der Übersetzung ermittelt und in entsprechender Weise gespeichert wird. Dadurch muß später nicht ständig wieder der gesamte Ausdruck zu Rate gezogen werden, was weniger effizient wäre.

Die Funktionen, die aus einem Ausdruck spezifische Informationen herausholen, sind nicht so kurz zu fassen, weswegen sie separat definiert werden mußten.

```
>   getConstructor = getTiExprConstructor
>   getSupercomb   = getTiExprSupercomb
>   getCase        = getTiExprCase
```

Das gilt auch für diejenigen, die Ausdrücke konstruieren, wie auch alle übrigen Funktionen der Klasse.

```
>   mkFreeVar      = mkTiExprFreeVar
>   mkApplication  = mkTiExprApplication
>   mkConstructor  = mkTiExprConstructor

>   applySubst     = applySubstTiExpr
>   reduce         = reduceTiExpr

>   fv             = fvTiExpr
>   fvNames        = fvTiExprNames

>   match          = matchTiExpr
```

Für die Funktionen, die aus einem Ausdruck bestimmte Komponenten isolieren, wird exemplarisch `getConstructor` erläutert.

```
> getTiExprConstructor tiExpr =
>   getConstructorFromTiState (TiExpr [] [])
>   . tiExpr2tiState tiExpr
```

Die Instanz für `TiExpr` wird also über `getConstructorFromTiState` ausgedrückt:

```
> getConstructorFromTiState :: (Addr -> a) -> TiState ->
>                                Maybe (Tag, Arity, [a])
> getConstructorFromTiState resFunc state@TiState { heap }
>   | exprType >= CWHNF = Just (tag, arity, args)
>   | otherwise         = Nothing
>   where
>     StateDesc { exprType,
>                 node, spine } = examineTiState state
>     (tag, arity, args) = case node of
>       NData tag arity args ->
```

```
>         (tag, arity,
>          map resFunc args)
>       NConstr tag arity    ->
>         (tag, arity,
>          map resFunc (take arity (getArgss heap spine)))
```

Diese arbeitet polymorph über dem Typ eines Ausdruckes und benutzt dazu eine Funktion `resFunc`, die aus den Argumentadressen wieder Ausdrücke erzeugt.

Nun wird am Beispiel von `mkTiExprApplication` demonstriert, daß die Konstruktion von Kernsprachenausdrücken recht einfach ist.

```
> mkTiExprApplication expr1 expr2 =
>   ST (mkNewTiExpr
>     (NAp (addrFromTiExpr expr1) (addrFromTiExpr expr2)))

> mkNewTiExpr node state@TiState { heap } =
>   (TiExpr [] [] new_addr, state { heap = new_heap })
>   where
>     (new_heap, new_addr) = hAlloc heap node
```

Die Anwendung von Substitutionen operiert im wesentlichen auf dem Heap, in dem die Ausdrücke mit Indirektionen auf das Substitut überschrieben werden.

```
> applySubstTiExpr subst's tiExpr =
>   foldM applyOneTiExprSubstTo tiExpr subst's
```

Die nicht angeführte Funktion `applyOneTiExprSubstTo` erledigt dies für jeweils eine einzelne Ersetzung[20] und gibt dabei einen `StateTransformer` zurück.

Beim Reduzieren schließlich sollen die Schritte der Template Instanziierungsmaschine übersprungen werden, die nur

- ein Abrollen des Spine darstellen,
- Indirektionen folgen oder
- Zwischenauswertungen für ein `case` starten

[20] D.h. einem Paar aus dem ursprünglichen und dem einzusetzenden Ausdruck

Dafür müssen aus der Folge der Zustände während der Reduktion einige herausgefiltert werden. Das macht die Funktion `reduceTiExpr` recht umfangreich, so daß auf ihre Darstellung verzichtet wird.

Auch die `match`-Funktion ist sehr umfangreich, weshalb sie nicht abgedruckt, sondern nur kurz ihre Vorgehensweise erläutert wird. Diese besteht darin, zu beiden Adressen die jeweiligen Knoten aus dem entsprechenden Heap zu ermitteln und im Falle einer Übereinstimmung rekursiv auf deren Komponenten fortzufahren.

5.5.4 Die Expansionsregeln

An den folgenden Ausführungen wird man sehen, daß sich der Aufwand gelohnt hat, von der konkreten Repräsentierung der Kernsprache zu abstrahieren. Denn dadurch haben wir relativ mächtige Funktionen zur Hand, um die Expansionsregeln schon fast so zu programmieren, wie sie in 5.2.1 formal beschrieben wurden.

Knoten im Tableau

Als Vorarbeit ist notwendig, einen Typen für den Inhalt eines Knotens im Tableau zu definieren.

```
> data (Eq a, CoreExpression e) => Label a e = Label {
>   ruleID      :: a,
>   revred      :: Maybe (Revred e), -- "Nothing" is solved!
>   env         :: Env,
>   subst's     :: Substitution e e,
>   loop        :: Maybe (Loop e),
>   addedFVs    :: [(e, Env)],
>   approxTerms :: [(e, Env)],
>   names       :: [Name]
> }

> data CoreExpression e => Loop e = Loop {
>   loopingNode :: Label RuleID e,
>   loopSubst   :: Substitution Name e
> }
```

Damit wird die `isLeaf`-Funktion nicht viel mehr sein als ein Pattern Match auf `Nothing`. Ein `revred` heißt aus historischen Gründen so, er speichert einfach eine der Bedingungen für die Kontextanalyse, wie sie in Abschnitt 5.1.1 erläutert wurden.

```
> data CoreExpression e => Revred e = (:<-:) {
>   expr :: e,
>   context :: ContextTerm
> } deriving (Eq)

> instance (CoreExpression e, Show e) => Show (Revred e) where
>   showsPrec x (expr :<-: context) =
>     showsPrec x expr .
>     showString " <- " .
>     showsPrec x context
>
```

Weil dort auch die Syntax von Anforderungstermen definiert wird, kann hier auf die Darstellung des Typs `ContextTerm` verzichtet werden.

Bei den Regeln sollen diejenigen an erster Stelle stehen, die speziell für die Gleichheitsanalyse implementiert werden mußten.

Für die Gleichheit spezielle Regeln

Diese erhalten zwar wie alle Expansionsregeln als Argument einen `Label`. Wichtig ist aber, daß sie explizit auf den Gleichheitsformeln arbeiten. Die Formeln sind im Prinzip durch Paare von Kernsprachenausdrücken repräsentiert:

```
> data CoreExpression e => EQRec e = EQRec {
>   left  :: e,
>   right :: e
> } deriving (Eq, Show)
```

Dieser Datentyp soll auch zu einer Instanz der Klasse `CoreExpression` gemacht werden, weil

- die allgemeinen Regeln für den Kontextkalkül nicht „wissen“ sollen, daß eigentlich sie auf Paaren von Ausdrücken operieren
- und dadurch an einigen Stellen recht kompakten Formulierungen möglich sind.

Die `is...`-Funktionen auf den Paaren stellen dabei hauptsächlich Boolesche Verknüpfungen der Funktionen auf den einzelnen Ausdrücken dar. Um den wesentlichen Gedanken davon zusammenzufassen, geben wir eine Funktion an, die die Kernsprachenprädikate auf `EQRec` erweitert:

```
> liftWith :: CoreExpression a =>
>             (a -> b -> c) -> (c -> c -> d) ->
>             EQRec a -> b -> d
> liftWith f binOp EQRec { left, right } =
>   binaryCombine binOp (f left) (f right)
```

Diese Definition mit Hilfe des Kombinators `binaryCombine`, der in Anhang B.1 zu finden ist, ist gleichwertig zu

```
 liftWith f binOp EQRec { left, right } env =
   (f left env) `binOp` (f right env)
```

Nun sind einige der Instanzenmethoden recht einfach anzugeben, weshalb gleich die Instanzendefinition folgt.

```
> instance CoreExpression e => CoreExpression (EQRec e) where
>   isWHNF   = isWHNF   `liftWith` (&&)
>   isFWHNF  = isFWHNF  `liftWith` (&&)
>   isCWHNF  = isCWHNF  `liftWith` (&&)
>   isSCWHNF = isSCWHNF `liftWith` (&&)
>   isPREDEX = isPREDEX `liftWith` (||)
>   isREDEX  = isREDEX  `liftWith` (||)

>   isFreeVar formula@EQRec { left, right } env =
>     isFreeVar left  env &&
>     isFreeVar right env && fvLeft == fvRight
>     where
>       Just fvLeft  = getFreeVar left  env
>       Just fvRight = getFreeVar right env
```

`getConstructor` und `getSupercomb` brauchen wir in dieser Instanz von `CoreExpression` nicht, daher geben wir beispielhaft nur `getCase` an:

```
>   getCase formula state =
>     (cases left) ++ (cases right)
>     where
>       cases side =
>         [ (expr', alts') |
>           (expr, alts) <- getCase (side formula) state,
```

```
>           let expr' = EQRec expr expr
>               alts' = map (liftAlt side) alts ]
>       liftAlt side (tag, arity, f) =
>         (tag, arity, f')
>         where
>           f' args = do
>             expr <- f (map side args)
>             return (EQRec expr expr)
```

Wie man sieht, werden einfach alle möglichen Programmkontexte für `case` von beiden Seiten der Formel zusammengenommen, wobei darauf geachtet werden muß, diese wieder zu `EQRec`'s zu machen.

Auch bei den Konstruktionsfunktionen läuft das nach diesem Schema ab, was man am Beispiel von `mkConstructor` nachvollziehen kann.

```
>   mkConstructor (tag, arity, args) = do
>     let
>       (largs, rargs) =
>         unzip [ (left, right) |
>                 EQRec { left, right } <- args ]
>     lconstr <- mkConstructor (tag, arity, largs)
>     rconstr <- mkConstructor (tag, arity, rargs)
>     return (EQRec lconstr rconstr)
```

Eine Substitution auf `EQRec` wird so aufgefaßt, daß linke durch linke und rechte durch rechte Seite ersetzt werden soll.

```
>   applySubst subst's formula@EQRec { left, right } =
>     do
>       lExpr <- applySubst lsubst's left
>       rExpr <- applySubst rsubst's right
>       return (formula { left = lExpr, right = rExpr })
>     where
>       (lsubst's, rsubst's) =
>         (unzip . map eqRec2expr) subst's
>       eqRec2expr (EQRec { left = l1, right = r1 } :->:
>                   EQRec { left = l2, right = r2 }) =
>         (l1 :->: l2, r1 :->: r2)
```

Dafür steht die lokale Funktion `eqRec2expr` bei der Anwendung einer Substitution. Bei `match` steht `expr2eqRec` nun für eine gleichmäßige Ersetzung links und rechts.

```
>   match (f1@(EQRec { left = l1, right = r1 }), env1)
>         (f2@(EQRec { left = l2, right = r2 }), env2) =
>     map (map expr2eqRec) (unionMaybeSubst matchr matchl)
>     where
>       matchl = match (l1, env1) (l2, env2)
>       matchr = match (r1, env1) (r2, env2)
>       expr2eqRec (x :->: y) =
>         (x :->: EQRec { left = y, right = y })
```

Nun kommen wir zu den Kalkülregeln. Bei der abstrakten Reduktion können wir zwei Ergebnisse erhalten, wenn es möglich ist, auf beiden Seiten zu reduzieren.

```
> eqAbstractReduction lab@Label { revred, env } =
>   maybe VNothing (listToVariousWith VOr . contracta) revred
>   where
>     contracta (expr@EQRec { left, right } :<-: context) =
>       [ lab { ruleID = AbsRedL,
>               revred =
>                 Just (expr { left = left' } :<-: context),
>               env    = env' } |
>         isREDEX left env,
>         (left', env') <- reduceN left env ] ++
>       [ lab { ruleID = AbsRedR,
>               revred =
>                 Just (expr { right = right' } :<-: context),
>               env    = env' } |
>         isREDEX right env,
>         (right', env') <- reduceN right env ]
```

Entsprechend markieren wir den Knoten mit dem jeweiligen Bezeichner für die Regel, um in der Heuristik später darauf zugreifen zu können. Außerdem sei erwähnt, daß durch `eqAbstractReduction` auch die Kalkülregel (CaseBot) implementiert wird, indem `case bot ...` von `reduce` auf `bot` „reduziert“ wird.

Nun folgt die Regel für das Hinzufügen von Argumenten, welche sich wiederum recht einfach darstellt.

```
> eqAddingArguments lab@Label { revred, env,
>                               names,  addedFVs } =
```

```
>   maybe VNothing adding revred
>   where
>     adding (expr@EQRec { left, right } :<-: context)
>       | (canAddArgumentTo `liftWith` (&&)) expr env =
>         let
>           ((newFreeVar, expr'), env') = (do
>             arg   <- mkFreeVar name
>             nexpr <- mkApplication expr arg
>             return (arg, nexpr)) `beginningWith` env
>         in
>         VJust lab {
>           ruleID   = AddArgs,
>           revred   = Just (expr' :<-: context),
>           env      = env',
>           addedFVs = (newFreeVar, env') : addedFVs,
>           names    = names' }
>       | otherwise  = VNothing
>       where
>         (name:names')   = names
>         canAddArgumentTo expr =
>           (isFWHNF expr) .||.
>           (isCWHNF expr .&&. (not . (isSCWHNF expr)))
```

Es wird eine neue Variable angelegt und der bestehende Ausdruck darauf angewendet, wenn er eine FWHNF ist. In der Komponente `addedFVs` werden hinzugefügte Variable gemerkt, damit die korrekte Lösung berechnet wird, indem diese Information von anderen Regeln abgefragt werden kann, vgl. dazu Abschnitt 5.2.1.

Bei der Approximation muß sichergestellt werden, daß es sich um die Anwendung eines `case` auf eine abstrakte Applikation handelt.

```
> eqApproximation lab@Label { revred, env, names, approxTerms } =
>  maybe VNothing approximating revred
>  where
>    approximating (expr@EQRec { left, right } :<-: context)
>      | (not . isREDEX expr .&&.
>         isCase expr .&&.
>         not . (isCase cexpr .||. isFreeVar cexpr)) env =
>        let
>          (expr', env') = (do
>            v <- mkFreeVar name
>            let
>              subst = cexpr :->: v
```

```
>           applySubst [subst] expr) `beginningWith` env
>       in
>       VJust lab {
>         ruleID      = Approx,
>         revred      = Just (expr' :<-: context),
>         env         = env',
>         approxTerms = (cexpr, env) : approxTerms,
>         names       = names' }
>     | otherwise = VNothing
>     where
>       isCase expr   = not . null . getCase expr
>       (cexpr, _)    = head (getCase expr env)
>       (name:names') = names
```

Dann kann diese durch eine neu eingeführte Variable ersetzt werden. Außerdem wird in der Komponente `approxTerms` der approximierte Term gespeichert, damit eine Anforderung an die neu eingeführte Variable korrekt umgesetzt werden kann.

Die Konstruktordekomposition unten erklärt sich weitgehend von selbst. Hervorgehoben sei das Zusammenspiel aus `isCWHNF` und der nur im Erfolgsfalle darauf folgenden Anwendung von `getConstructor`.

```
> eqConstructorDecomposition lab@Label { revred, env } =
>   maybe VNothing (listToVariousWith VAnd . decompose) revred
>   where
>     decompose (expr@EQRec { left, right } :<-: context)
>       | isCWHNF expr env
>         && (tagl == tagr) && (arityl == arityr) =
>         [ lab { ruleID = CDecomp,
>                 revred = Just
>                   (EQRec { left, right } :<-: context) } |
>           (left, right) <- zip argsl argsr ]
>       | otherwise = []
>       where
>         Just (tagl, arityl, argsl) =
>           getConstructor left env
>         Just (tagr, arityr, argsr) =
>           getConstructor right env
```

Der Vergleich von Konstruktor und Stelligkeit erfolgt auf die gleiche Weise wie bei der Funktionsdekomposition. Trotzdem sieht diese doch ein wenig anders aus. Das liegt daran, daß wir in der Klasse `CoreExpression` keine

Funktion definiert haben, die uns darüber Auskunft gibt, ob wir eine Anwendung eines Superkobinators vor uns haben. Man hätte das sicherlich integrieren können; auf der anderen Seite war die Lösung über `getSupercomb` und die Abfrage, ob es sich um `Just` handelt, schnell gefunden und ist genauso brauchbar.

```
> eqFunctionDecomposition lab@Label { revred, env } =
>   maybe VNothing (listToVariousWith VAnd . decompose) revred
>   where
>     decompose (expr@EQRec { left, right } :<-: context) =
>       [ lab { ruleID = FDecomp,
>               revred = Just
>                 (EQRec { left, right } :<-: context) } |
>           let supercombl = getSupercomb left env
>               supercombr = getSupercomb right env,
>           isJust supercombl, isJust supercombr,
>           let Just (namel, arityl, argsl) = supercombl
>               Just (namer, arityr, argsr) = supercombr,
>           namel == namer, arityl == arityr,
>           (left, right) <- zip argsl argsr ]
```

Stellt die Regel `eqSyntacticallyEqual` syntaktische Gleichheit der beiden Ausdrücke fest, so kann die Anforderung entfernt werden, da wir es ja mit dem trivialen Fall zu tun haben.

```
> eqSyntacticallyEqual lab@Label { revred, env } =
>   maybe VNothing solving revred
>   where
>     solving (EQRec { left, right } :<-: CConstr 1 0 [])
>       | syntEq left right env =
>         VJust lab { ruleID = SyntEQ,
>                     revred = Nothing }
>       | otherwise     = VNothing
>     solving _ =
>       error ("EQLocalRules.eqSyntacticallyEqual: "
>                ++ "Context is not <True> for EQ-Tableau!")
```

Die Regel `eqBindFreeVariable` versucht eine Substitution auf den freien Variablen zu finden, die die beiden Ausdrücke zur Deckung bringt. Hierbei darf keine zyklische Substitution berechnet werden, was die Funktion `match` sicherzustellen hat.

```
> eqBindFreeVariable lab@Label { revred, env } =
>   maybe VNothing (listToVariousWith VOr . binding) revred
>   where
>     binding (expr@EQRec { left, right } :<-: context) =
>       maybeToList (makeBinding left right) ++
>       maybeToList (makeBinding right left)
>       where
>         makeBinding from to =
>           maybe Nothing
>             makeNewLabel
>             (map
>               (map nameSubst2EQRecExprSubst)
>               (match (from, env) (to, env)))
```

eqBindFreeVariable ist etwas umfangreicher. Das obige Fragment genügt aber sicherlich, die prinzipielle Vorgehensweise deutlich zu machen.

Allgemeine Regeln

Nun kommen die Regeln an die Reihe, die allgemein für die Kontextanalyse verwendbar sind.

Die abstrakte Reduktion wäre eigentlich ein Kandidat hierfür gewesen, wäre sie nicht schon auf EQRec's implementiert, um die Auswahl zwischen einer Reduktion links oder rechts zu ermöglichen. Also bleibt schlußendlich nur noch die Regel caseFreeVariable übrig.

```
> caseFreeVariable lab@Label { revred, env, subst's, names } =
>   maybe VNothing (listToVariousWith VOr . casing lab) revred
```

Da bot intern als eine freie Variable dargestellt wird, muß dieser Fall hier ausgeschlossen werden.

```
> casing lab@Label { revred, env, subst's, names, addedFVs }
>        (expr :<-: context)
>   | isPREDEX expr env &&
>     name `notElem` addedFVnames &&
>     name /= "bot"  = botNode : do
>       (tag, arity, _) <- alts
>       let
>         (vs, names') = splitAt arity names
>         ((subst, expr'), env') = (do
```

```
>           args <- mapM mkFreeVar vs
>           sexp <- mkConstructor (tag, arity, args)
>           let
>             subst = v :->: sexp
>           nexp <- applySubst [subst] expr
>           return (subst, nexp)) `beginningWith` env
>       return lab { ruleID  = CaseFV,
>                    revred  = Just (expr' :<-: context),
>                    env     = env',
>                    subst's = subst : subst's,
>                    names   = names' }
>   | otherwise      = []
>   where
>     (v, alts)      =
>       head [ (t, as) |
>              (t, as) <- getCase expr env,
>              isFreeVar t env ]
>     Just name      = getFreeVar v env
>     addedFVnames   = [ name |
>       Just name <- map (uncurry getFreeVar) addedFVs ]
```

Die lokale Definition für `(v, alts)` bedarf einer Erläuterung. Da abgefragt wird, ob der Ausdruck ein potentieller Redex ist, kann die List Comprehension auf deren rechter Seite nicht leer sein, und somit ist die Anwendung von `head` immer wohldefiniert.

Daneben sei darauf hingewiesen, wie die Forderung, daß das `case` vollständig spezifiziert sein muß, die Bildung der entsprechenden Konstruktorausdrücke ermöglicht bzw. vereinfacht. Ohne diese Forderung müßten wir in der Umgebung Informationen über alle verwendeten Konstruktoren mitführen.

```
>     botNode        =
>       let
>         ((subst, expr'), env') = (do
>           botVar <- mkFreeVar "bot"
>           let
>             subst = v :->: botVar
>           nexp <- applySubst [subst] expr
>           return (subst, nexp)) `beginningWith` env
>       in lab { ruleID  = CaseFV,
>                revred  = Just (expr' :<-: context),
>                env     = env',
>                subst's = subst : subst's }
```

Wie man sieht, muß die spezielle Alternative für `bot` separat definiert werden, da es sich nicht um eine Konstruktoranwendung handelt.

5.5.5 Gleichheitstableau

Die eigentliche Analyse samt der Gleichheitstableau ist in einem Modul namens APE[21] untergebracht.

```
> module APE where
```

Ein Menge von Gleichheitstableaus definieren wir durch ein `TableauSet` mit dem entsprechenden Knoteninhaltstyp. Dieser ist ein `Label` mit einer `RuleID` und einem `EQRec`.

```
> type EQTableauSet e = TableauSet (Label RuleID (EQRec e))
```

Außerdem brauchen wir einen Typ für ein einzelnes Tableau. Ein solches werden wir nämlich nach der Anwendung der Heuristik vor uns haben. Ein Gleichheitstableau wird dann entsprechend definiert.

```
> data Tableau a = TBranch (Various (Tableau a)) a
>                 | TLeaf a

> type EQTableau e = Tableau (Label RuleID (EQRec e))
```

Ein solches Tableau kann potentiell unendlich sein; daher möchten wir eine Funktion, die es nach einer gegebenen Tiefe abschneidet.

```
> takeTableau depth tableau@(TLeaf _) =
>   tableau

> takeTableau depth tableau@(TBranch succ lab)
>    | depth == 0 = TLeaf lab
>    | otherwise  =
>      TBranch (map (takeTableau (depth -1)) succ) lab
```

[21] engl. *analyser for program equivalence*

Nun können wir eine Funktion angeben, die eine Menge von Gleichheitstableaus erstellt aus einem Starttableau und all den Expansionsregeln, die wir in Abschnitt 5.2.1 vorgestellt haben.

```
> makeEQTableau :: (Show e, CoreExpression e) =>
>                  EQTableauSet e -> EQTableauSet e
> makeEQTableau =
>    expandTableau finalEQNode expansionRules

> expansionRules :: (Show expr, CoreExpression expr) =>
>                   [ExpansionRule (Label RuleID (EQRec expr))]
> expansionRules = [ eqSyntacticallyEqual,
>                    eqConstructorDecomposition,
>                    eqFunctionDecomposition,
>                    eqAbstractReduction,
>                    caseFreeVariable,
>                    eqBindFreeVariable,
>                    eqAddingArguments,
>                    eqApproximation
>                  ]
```

An einem Knoten kann dabei nicht mehr weiter expandiert werden, wenn dort die Anforderung schon gelöst und damit gelöscht wurde.

```
> finalEQNode lab@Label { revred } =
>   (not . isJust) revred
```

Das Schließen eines `EQTableauSet` besteht darin, mögliche Schleifen zu finden.

```
> closeEQTableau :: CoreExpression e =>
>                   EQTableauSet e -> EQTableauSet e
> closeEQTableau =
>   loopTableau matchLabels rootConditions pathConditions
```

Dabei ist `matchLabels` eine Funktion, die ermittelt, ob die zwei Knoten aufeinander abgebildet werden können. `rootConditions` und `pathConditions` sind die Bedingungen an den Pfad von der Wurzel bzw. zwischen den beiden Knoten, die eine Schleife bilden sollen, wie sie in Abschnitt 5.2.2 spezifiziert wurden.

5.5.6 Lösungen

Nun geht es darum, für die in einem Gleichheitstableau implizit steckende Lösung eine explizite Darstellung in Form einer Anforderung zu finden. Dazu werden Lösungen für einzelne Terme ähnlich zu den Anforderungen definiert.

```
> data SimpleSolution = Bot
>                     | Top
>                     | Var Name
>                     | Constr Tag Arity [SimpleSolution]
>                     | Ap SimpleSolution SimpleSolution
>                     deriving (Eq)
```

Außerdem wird eine Klasse von Termen erklärt, für die man eine Lösung in der o.g. Form berechnen kann.

```
> class Solvable t where
>   simpleSolution :: t -> Env -> SimpleSolution
```

`TiExpr` und `EQRec` werden dann auf naheliegende Weise zu Instanzen dieser Klasse gemacht.

```
> instance Solvable TiExpr where
>   simpleSolution tiExpr =
>     simpleAddrSolution (addrFromTiExpr tiExpr)

> instance (CoreExpression e, Solvable e) =>
>          Solvable (EQRec e) where
>   simpleSolution EQRec { left } =
>     simpleSolution left

> simpleAddrSolution :: Addr -> Env -> SimpleSolution
> simpleAddrSolution addr (state@TiState { heap }) =
>   makeFromNode (hLookup heap addr)
>   where
>     makeFromNode (NAp a1 a2) =
>       Ap (simpleAddrSolution a1 state)
>          (simpleAddrSolution a2 state)
>     makeFromNode (NData tag arity args) =
>       Constr
>         tag
```

```
>         arity
>         (map (flip simpleAddrSolution state) args)
>     makeFromNode (NFVar name)
>       | name == "bot" = Bot
>       | otherwise     = Var name
>     makeFromNode (NInd addr) =
>       makeFromNode (hLookup heap addr)
>     makeFromNode _ =
>       error "APE.simpleAddrSolution: no simple term"
```

Nun geht es darum, Lösungen zu erklären, die nur gemeinsam gelten.

```
> data JointSolution
>   = NoSolution
>   | Context Name
>   | Tuple [SimpleSolution]
>   | JointSolution `Where`
>       ([SimpleSolution], JointSolution)
>   | Union [JointSolution]
>   | Intersection [JointSolution]
>   | JointSolution `Needs` [(SimpleSolution, SimpleSolution)]
>   deriving (Eq)
```

Mit Hilfe dieser Datentypen und Funktionen kann `solution` definiert werden, welche aus einem Tableau die Lösungsanforderung generiert. Das Vorgehen entspricht dabei exakt den Ausführungen aus Abschnitt 5.3.1, weswegen hier nur der Typ von `solution` angegeben wird.

```
> solution :: (CoreExpression expr, Solvable expr) =>
>             Name ->
>             Tableau (Label RuleID expr) ->
>             (Name, [Name], JointSolution)
```

5.5.7 Heuristik

Die Heuristik ist an sich nicht kompliziert, die Funktion wegen einer Reihe von Fallunterscheidungen aber nicht mehr so übersichtlich, so daß hier nur kurz das prinzipielle Vorgehen verdeutlicht werden soll. Können mehrere Regeln erfolgreich angewendet werden, so wird der Reihenfolge nach diejenige ausgewählt, die

1. ein Blatt hervorbringt, d.h. die Anforderung des Knotens löst.
2. eine abstrakte Reduktion durchführt. Ist links *und* rechts eine Reduktion möglich, so wird die Seite ausgewählt, auf der nicht zuletzt schon reduziert wurde. Zum Speichern dieser Information dient der Datentyp

   ```
   > data Info a = Info { lastRed :: a }
   ```

3. in der durch die Reihenfolge der Konstruktoren bei der Definition des `RuleID`-Typs vorgegebenen Ordnung die größte ist. Im speziellen bedeutet das, daß eine Funktionsdekomposition nur dann angewendet wird, wenn alle anderen Regeln ausscheiden.

Im Übrigen ist die Möglichkeit zur Anwendung einer Regel stark durch der Struktur der Terme bestimmt; z.B. werden die Regeln (AddArgs) und (CaseFV) schlecht gleichzeitig zur Wahl stehen können.

5.5.8 Erstellen des initialen `TableauSet`

Nun geht es darum, wie das initiale `TableauSet` aufgebaut wird, und auf welche Arten die Analyse benutzt werden kann. Der Startknoten bekommt auch eine `RuleID`, nämlich `Initial` versehen mit einem Namen, der später auch als Name der Lösung verwendet wird.

```
> initialNode initialName (formula, initialEnv) =
>   TSBranch [] Label {
>     ruleID      = Initial initialName,
>     revred      = Just (formula :<-: CConstr 1 0 []),
>     env         = initialEnv,
>     subst's     = [],
>     loop        = Nothing,
>     addedFVs    = [],
>     approxTerms = [],
>     names       = newNames formula initialEnv
>   }
```

Als neue Namen für freie Variable dürfen in dem Tableau nur solche verwendet werden, die nicht ursprünglich in der Formel auftauchen.

```
> newNames expr env =
>   filter
```

```
>     (`notElem` (fvNames expr env))
>     (map (\ n -> 'x' : show n) [1..])

> newName (x:xs) = (x, xs)
> newName _      = undefined
```

Um ein initiales `EQTableauSet` zu erstellen, muß die Funktion `initialNode` mit den entsprechenden Argumenten aufgerufen werden. Dies erledigt die Funktion `prepAnalyse`, die aus einem Quellprogramm und einer Liste von Formeln eine Liste von `EQTableauSet`s erzeugt.

Das Quellprogramm ist dabei als Zeichenkette gegeben, wie auch die Ausdrücke in den Formeln, die als Paare repräsentiert werden. Daraus erklärt sich der Typ der Funktion,

```
> prepAnalyse :: String -> [(String, String)] ->
>                [EQTableauSet TiExpr]
> prepAnalyse source formulas =
>   let
>     lrSides  =
>       [ ((lhsL, rhsL), (lhsR, rhsR))
>         | ((rhsL, rhsR), number)
>           <- zip formulas (map show [1..]),
>          let lhsL = "mainL" ++ number,
>          let lhsR = "mainR" ++ number ]
>     exprDefs =
>       concat [ lhsL ++ " = " ++ rhsL ++ ";\n\n" ++
>                lhsR ++ " = " ++ rhsR ++ ";\n\n"
>                | ((lhsL, rhsL), (lhsR, rhsR)) <- lrSides ]
>     initialEnv =
>       (compile . parse)
>         (source ++ "\n\n" ++ exprDefs)
>     bindings   =
>       (map (\ (x,y,_) -> (x,y)) . globals) initialEnv
>     exprAddrs  =
>       [ ((rhsL, addrL), (rhsR, addrR))
>         | ((lhsL, rhsL), (lhsR, rhsR)) <- lrSides,
>          let addrL = (fromJust . flip lookup bindings) lhsL,
>          let addrR = (fromJust . flip lookup bindings) lhsR ]
>   in
>     [ initialNode
>         (rhsL ++ " " ++ formulaToken ++ " " ++ rhsR)
>         ((do
>            exprL <- reduce (TiExpr [] [] addrL)
```

```
>              exprR <- reduce (TiExpr [] [] addrR)
>              return (EQRec exprL exprR))
>            `beginningWith` initialEnv)
>        | ((rhsL, addrL), (rhsR, addrR)) <- exprAddrs ]
```

die folgendermaßen vorgeht:

- Zu jeder Formel der Art `(rhsL, rhsR)`, werden Namen `mainL` und `mainR` für Superkombinatoren erzeugt und mit der jeweiligen Nummer der Formel versehen. Dies geschieht in `lrSides`.
- Nun stehen zu linkem und rechtem Ausdruck jeder Formel jeweils eine linke und eine rechte Seite für eine Superkombinatordefinition zur Verfügung, was in `exprDefs` syntaktisch korrekt erledigt wird.
- Das Quellprogramm wird nun zusammen mit den zusätzlichen Superkombinatordefinitionen übersetzt, was einen Anfangszustand `initialEnv` der Template Instanziierungsmaschine hervorbringt.
- Aus diesem Anfangszustand müssen wiederum die Adressen der für die zu untersuchenden Ausdrücke definierten Superkombinatoren herausgesucht werden. Diese werden zusammen mit der textuellen Repräsentation der ursprünglichen Ausdrücke in `exprAddrs` gespeichert.
- Schließlich erfolgt der Aufruf von `initialNode` mit einem Namen, der die Formel textlich darstellt, und einem Paar aus der Formel als `EQRec` von `TiExpr` und der Umgebung. Damit auf Anhieb die gewünschten Ausdrücke im ersten Tableauknoten landen und nicht die hierfür definierten Superkombinatoren, wird eigens ein erster Reduktionsschritt ausgeführt.

Als Begrenzer zwischen linker und rechter Seite der Formel wurde

```
> formulaToken = "=?"
```

gewählt. Dieser wird auch in einer Funktion `processFormulas` benutzt, die eine Reihe von `=?`-Formeln aus einer Datei einliest. Dabei muß jede Zeile entweder eine Formel enthalten, leer sein oder mit `--` beginnen.

```
> processFormulas fileContents =
>   let
>     formulas = lines fileContents
>   in
```

```
>     [ (concat (intersperse " " lhs),
>        concat (intersperse " " rhs))
>       | line <- formulas,
>         let lexedLine = words line,
>         (not . null) lexedLine &&
>         (not ("--" `isPrefixOf` (head lexedLine))),
>         let (lhs, x:rhs) = span (/= formulaToken) lexedLine ]
```

Die Funktion, mit der man die Analyse durchführen kann, setzt sich dann ganz einfach zusammen.

```
> analyse source =
>   map analyseFunc . prepAnalyse source

> analyseFunc :: (Show expr, CoreExpression expr) =>
>                EQTableauSet expr -> EQTableau expr
> analyseFunc =
>   heuristic Info { lastRed = 0 }
>    . closeEQTableau
>    . makeEQTableau
```

Dabei hat man dann immer noch die Wahl, die Formeln direkt oder — indem man den Aufruf von `processFormulas` vorschaltet — in einer Datei anzugeben. Die beschriebenen Funktionen können von einem anderen Programm oder bequem vom Interpreter `hugs` aus aufgerufen werden. Somit läßt sich die Gleichheitsanalyse bereits sinnvoll durchführen.

5.5.9 Das Hauptmodul für die Analyse

Damit die Analyse aber auch als selbständig ausführbares Programm vorliegt, gibt es ein Hauptmodul. Dieses interpretiert die Kommandozeile, um die Gleichheitsanalyse mit Hilfe der in Abschnitt 5.5.8 vorgestellten Funktionen durchzuführen. Die Syntax für einen Programmaufruf lautet dann

```
ape [-options] [source [formulas]]
```

Hierbei können die Formeln in einer Datei abgelegt sein, die die Form aufweist, wie sie zuvor für `processFormulas` beschrieben wurde. Gibt man keinen Namen für eine solche Datei an, so fordert das Programm interaktiv zur Eingabe einer Formel auf. Läßt man auch die Angabe einer Quelldatei, welche das Kernsprachenprogramm beinhaltet, fort, wird deren Name vom Programm erfragt. Die möglichen Optionen umfassen

-depth=n für die maximale Tiefe, die das Tableau erreichen darf.

-disp für die Ausgabe des Tableaus anstelle der Lösungsanforderung

5.6 Durchführung

In diesem Abschnitt wird kurz auf organisatorische Aspekte bei der Durchführung der Implementierung eingegangen, und es werden einige wichtige Werkzeuge, die dabei zum Einsatz kamen, genannt.

5.6.1 Organisation

Wie schon einmal angesprochen, wurden allgemein verwendbare Funktionen in Bibliotheks-Modulen untergebracht. Diese Module finden ihren Platz allesamt in einem Unterverzeichnis names `lib`. Ein weiteres Verzeichnis `template-instantiation` dient dazu, die Module aus der Template Instanziierungsmaschine aufzunehmen. Für die Gleichheitsanalyse spezifische Module und die betreffenden Dokumente sind in `program-equivalence` zu finden. Teile, die allgemein für die Kontextanalyse eingesetzt werden können, wurden in dem Verzeichnis `evaluation-contexts` abgelegt. Darüber hinaus gibt es noch ein Verzeichnis `samples`, welches verschiedene Formeldateien und Kernsprachenprogramme aufnimmt, darunter auch die für die Analysebeispiele in Anhang A.

5.6.2 Werkzeuge

Als Versionskontrollsystem bot sich `cvs`[22] an, da es fähig ist, auch eine Verzeichnisstruktur, wie die oben beschriebene zu verwalten. Der `hbc`[23] wurde zur Erzeugung des ausführbaren Analyseprogramms eingesetzt. Um während der Entwicklung Teile des Programmes zu testen, war jedoch der schon in Abschnitt 5.5.8 erwähnte Interpreter `hugs`[24] wesentlich hilfreicher.

[22] Concurrent Versions System
[23] Chalmers Haskell B Compiler der Version 0.9999.4
[24] Hugs 1.4, The Nottingham and Yale Haskell User's System

Kapitel 6

Ergebnisse

In diesem Kapitel geht es um die Resultate, die mit der beschriebenen Implementierung erreicht wurden. Diese werden unter zweierlei Gesichtspunkten betrachtet. Der erste ist der qualitative und befaßt sich damit, welche Formeln bewiesen werden können und wo die Grenzen liegen. Dies wird anhand von Beispielen dargelegt.

Der zweite Aspekt ist die Leistungsfähigkeit im Hinblick darauf, daß die Analyse praktikabel einsetzbar ist.

6.1 Qualitative Resultate: Mächtigkeit anhand von Beispielen

Mit dem Programm lassen sich die meisten in [SS96a] angeführten Beispiele beweisen.

6.1.1 Erste Prüfsteine

Diese bildeten die ersten Prüfsteine. Dazu werden folgende Definitionen vorausgesetzt:

```
--
-- Simple Combinators
--

id x = x;

compose f g x = f (g x);

K  x y = x;
K1 x y = y;

Y f = f (Y f);
```

```
D f x = f (x x);
F f   = (D f) (D f);

Nil         = Cons{1,0};
Kons   x xs = Cons{2,2} x xs;

True  = Cons{3,0};
False = Cons{4,0};
```

Die für bestimmte Konstruktoren gewählten Synonyme werden der Einfachheit halber auch bei der Angabe der Lösung verwendet. Bei den Beispielen soll die Numerierung aus o.g. Papier beibehalten werden.

cpe4: Die Gleichheit von `Kons` und `Cons{2,2}` gilt ohne Einschränkungen.

cpe5: Für `f = f (f x)` und `g x = g x` kann das Programm die Gleichheit *nicht* beweisen, da die Approximationsregel in dieser Arbeit von speziellerer Natur ist als in [SS96a], vgl. dazu Abschnitt 5.2.1.

cpe6: Aber die Gleichheit der Fixpunktkombinatoren `Y` und `F` kann gezeigt werden.

cpe7: Die Terme `case x of <1> -> Nil; <2> x xs -> Kons x xs` und `id x` sind für die Lösungsanforderung $x = \langle \texttt{Bot}, \texttt{Nil}, \texttt{Kons Top Top} \rangle$ äquivalent. Hierbei ist interessant, daß diese Lösungsanforderung für x eine echte Obermenge von $List = \langle \texttt{Bot}, \texttt{Nil}, \texttt{Kons Top } List \rangle$ ist.

Verwendet man hingegen die folgende Funktion `idOnList`,

```
idOnList xs = case xs of
  <1>      -> Nil;
  <2> x xs -> Kons x (idOnList xs);
```

die eine rekursive Identität auf Listen darstellt, so erhält man auf die Anfrage `idOnList` $xs \doteq$ `id` xs hin Gleichheit für die Anforderung $xs = \langle \texttt{Bot}, \texttt{Nil}, \texttt{Kons Top } x2 \texttt{ where } x2 = xs \rangle$, welche nach [Sch, Lemma 2.11] mit $List$ übereinstimmt. Für das folgende sind noch einige weitere Funktionsdefinitionen nötig.

```
--
-- Functions on Lists
--

append xs ys = case xs of
  <1>      -> ys;
```

```
  <2> x xs -> Cons{2,2} x (append xs ys);

filter p as = case as of
  <1>      -> Nil;
  <2> x xs -> if (p x) (Kons x (filter p xs)) (filter p xs);

map f as = case as of
  <1>      -> Nil;
  <2> x xs -> Kons (f x) (map f xs);

rev as = case as of
  <1>      -> Nil;
  <2> x xs -> append (rev xs) (Kons x Nil);

length as = case as of
  <1>      -> Zero;
  <2> x xs -> Succ (length xs);

--
-- Functions on Booleans
--

if pred then else = case pred of
  <3> -> then;
  <4> -> else;
```

Nun können die übrigen Beispiele aus [SS96a] behandelt werden.

cpe8: Für die eben gegebenen Definitionen von `map` und `filter` kann die Äquivalenz von `filter p (map f l)` und `map f (filter (compose p f) l)` gezeigt werden. Wir erhalten dabei in der Lösung zusätzliche Anforderungen an den approximierten Term.

$$
\begin{array}{ll}
\multicolumn{2}{l}{MapFilterEQ \;=} \\
\langle & (p, f, \texttt{Bot}), (p, f, \texttt{Nil}), \\
 & (p, f, \texttt{Kons}\ x1\ \texttt{Top})\ \texttt{needs}\ p\ (f\ x1)\ \in\ \texttt{Bot}, \\
 & (p, f, \texttt{Kons}\ x1\ x2)\ \texttt{where} \\
 & (p, f, x2)\ =\ MapFilterEQ\ \texttt{needs}\ p\ (f\ x1)\ \in\ \langle\texttt{True}\rangle \\
 & (p, f, \texttt{Kons}\ x1\ x2)\ \texttt{where} \\
 & (p, f, x2)\ =\ MapFilterEQ\ \texttt{needs}\ p\ (f\ x1)\ \in\ \langle\texttt{False}\rangle\ \rangle
\end{array}
$$

Interessant ist daran, daß diese Äquivalenz dem *take-lemma* (siehe [BW88, S. 182ff.]) nicht ohne weiteres zugänglich ist, d.h. nur mit zusätzlichen Annahmen gezeigt werden kann (vgl. [Gor94]).

cpe9: Die Gleichheit von `length (rev xs)` und `length xs` kann nicht nachgewiesen werden. Man erhält als Lösung nur die trivialen Fälle $xs = \langle \texttt{Bot}, \texttt{Nil} \rangle$.

6.1.2 Beispiele mit Peano-Zahlen

Darüber hinaus gibt es noch einige interessante Beispiele, die mit Peano-Zahlen zu tun haben. Dazu werden diese und die Additionsfunktion in der Kernsprache definiert.

```
--
-- Functions on Numbers
--

Zero = Cons{5,0};
Succ x = Cons{6,1} x;

add x y = case x of
  <5>    -> y;
  <6> z -> Cons{6,1} (add z y);
```

peano1: `add x Zero` ist äquivalent zu `x`. Dies ergibt die Lösungsanforderung $Num = \langle \texttt{Bot}, \texttt{Zero}, \texttt{Succ}\ x\ \texttt{where}\ x = Num \rangle$.

peano2: Der Versuch, die Gleichheit von `add x (Succ y)` und `Succ (add x y)` zu zeigen, führt leider nicht zu einem geschlossenen Tableau, und damit auch nicht zu einer geschlossenen Form der Lösungsanforderung. Je nachdem, wie tief man das Tableau auswerten läßt, kann man z.B. folgende Lösungen erhalten:

(11 Schritte)	$(x, y) = \langle (\texttt{Zero}, \texttt{Top}) \rangle$
(19 Schritte)	$(x, y) = \langle (\texttt{Zero}, \texttt{Top}), (\texttt{Succ Zero}, \texttt{Top}) \rangle$
(27 Schritte)	$(x, y) = \langle (\texttt{Zero}, \texttt{Top}), (\texttt{Succ Zero}, \texttt{Top}),$ $(\texttt{Succ (Succ Zero)}, \texttt{Top}) \rangle$

peano3: Die Assoziativität von `add` kann gezeigt werden, d.h. für die Lösungsanforderung

$$\begin{aligned} AddAssoc = \langle & (\texttt{Bot}, \texttt{Top}, \texttt{Top}), \\ & (\texttt{Zero}, \texttt{Bot}, \texttt{Top}), (\texttt{Zero}, \texttt{Zero}, \texttt{Top}), (\texttt{Zero}, \texttt{Succ Top}, \texttt{Top}), \\ & (\texttt{Succ}\ x, y, z)\ \texttt{where}\ (x, y, z) = AddAssoc \rangle \end{aligned}$$

ist `add x (add y z)` äquivalent zu `add (add x y) z`.

6.1.3 Weitere Beispiele auf Listen

Es gilt, noch weitere Beispiele auf Listen zu untersuchen.

lists1: `append xs Nil` ist äquivalent zu `xs` mit der Lösungsanforderung $List = \langle \texttt{Bot}, \texttt{Nil}, \texttt{Kons Top}\ xs\ \texttt{where}\ xs = List\rangle$. Man beachte, daß die Form der Lösung der von Beispiel **peano1** gleicht.

lists2: Die Assoziativität von `append` kann gezeigt werden; das bedeutet, `append xs (append ys zs)` ist gleich zu `append (append xs ys) zs` für die Anforderung:

$$\begin{aligned} AppAssoc = \ & \langle (\texttt{Bot}, \texttt{Top}, \texttt{Top}), \\ & (\texttt{Nil}, \texttt{Bot}, \texttt{Top}), (\texttt{Nil}, \texttt{Nil}, \texttt{Top}), (\texttt{Nil}, \texttt{Kons Top Top}, \texttt{Top}), \\ & (\texttt{Kons Top}\ xs, ys, zs)\ \texttt{where}\ (xs, ys, zs) = AppAssoc\rangle \end{aligned}$$

Wiederum ist die Ähnlichkeit zum Beispiel **peano3** nicht zu übersehen.

lists3: Interessant ist wohl auch die Äquivalenz von `length (append xs ys)` und `add (length xs) (length ys)`, die für

$$\begin{aligned} AppLengthEQ = \ & \langle (\texttt{Bot}, \texttt{Top}), \\ & (\texttt{Nil}, \texttt{Bot}), (\texttt{Nil}, \texttt{Nil}), (\texttt{Nil}, \texttt{Kons Top Top}), \\ & (\texttt{Kons Top}\ xs, ys)\ \texttt{where}\ (xs, ys) = AppLengthEQ\rangle \end{aligned}$$

gezeigt werden kann.

lists4: Auf die Anfrage $\texttt{append}\ xs\ ys \ \doteq\ xs$ hin liefert das Programm die Lösung

$$\begin{aligned} AppEQ = \ & \langle (\texttt{Bot}, \texttt{Top}), (\texttt{Nil}, \texttt{Nil}), \\ & (\texttt{Kons Top}\ xs, ys)\ \texttt{where}\ (xs, ys) = AppEQ\rangle, \end{aligned}$$

die somit die von Beispiel **lists1** beinhaltet.

Eine Funktion, die eine unendliche Liste mit ihrem Argument füllt, kann man auf verschiedene Arten angeben.

```
repeat  x = Kons x (repeat x);
repeatY x = Y (Kons x);
repeat2 x = Kons x (Kons x (repeat2 x));
repeat3 x = Kons x (Kons x (Kons x (repeat3 x)));
```

lists5: All diese verschiedenen Definitionen für `repeat` können ohne Einschränkungen an das Argument x als gleich bewiesen werden.

6.1.4 Beispiele mit Booleschen Funktionen

Eine weitere Gruppe bilden Untersuchungen bezüglich der logischen Funktionen:

```
--
-- Functions on Booleans
--

or x y = case x of
 <3> -> True;
 <4> -> y;

and x y = case x of
 <3> -> y;
 <4> -> False;
```

bools1: Die Symmetrie von `or` kann *selbstverständlich nicht* in allen Fällen bewiesen werden, da

1. die gegebene Definition von `or` nicht in beiden, sondern nur im ersten Argument strikt ist.
2. wir auf der anderen Seite mit den vorhandenen Sprachmitteln kein *paralleles Oder*[1] darstellen können.

So ergibt sich für die Anfrage `or x y` $\doteq$ `or y x` die Lösungsanforderung

$$\begin{aligned} OrSymm = \ \langle & (\texttt{Bot},\texttt{Bot}), (\texttt{Bot},\texttt{False}), \\ & (\texttt{True},\texttt{True}), (\texttt{True},\texttt{False}), \\ & (\texttt{False},\texttt{Bot}), (\texttt{False},\texttt{True}), (\texttt{False},\texttt{False})\rangle \end{aligned}$$

Wie man sieht, „fehlen“ gerade die Fälle (`Bot`, `True`) und (`True`, `Bot`), da im ersten zu `bot` und im zweiten zu `True` reduziert wird.

bools2: Für die `and`-Funktion sieht es nicht anders aus. In der Lösungsanforderung sind lediglich die Paare (`Bot`, `False`) und (`False`, `Bot`) nicht vertreten:

$$\begin{aligned} AndSymm = \ \langle & (\texttt{Bot},\texttt{Bot}), (\texttt{Bot},\texttt{True}), \\ & (\texttt{True},\texttt{Bot}), (\texttt{True},\texttt{True}), (\texttt{True},\texttt{False}), \\ & (\texttt{False},\texttt{True}), (\texttt{False},\texttt{False})\rangle \end{aligned}$$

[1] D.h., es läßt sich in der Kernsprache kein Superkombinator `parOr` angeben, so daß sowohl `parOr bot True` als auch `parOr True bot` zu `True` reduzieren würde.

Betrachtet man die Versionen der Funktionen `or` resp. `and`, die in beiden Argumenten strikt sind, so ist die Lösungsanforderung für die Symmetrie jeweils $(Bool, Bool)$ mit $Bool = \langle \texttt{Bot}, \texttt{True}, \texttt{False} \rangle$. Für die folgenden Funktionen `strictOr` bzw. `strictAnd`

```
strictOr x y = case x of
  <3> -> (case y of
    <3> -> True;
    <4> -> True);
  <4> -> (case y of
    <3> -> True;
    <4> -> False);

strictAnd x y = case x of
  <3> -> (case y of
    <3> -> True;
    <4> -> False);
  <4> -> (case y of
    <3> -> False;
    <4> -> False);
```

kann also Symmetrie gezeigt werden. Aber auch wenn Striktheitsinformationen beim Übersetzungsprozess ausgenutzt werden können (vgl. [Sch94]), ist es nicht generell wünschenswert, möglichst strikte Funktionen anzugeben. Denn dies widerspräche dem Charakter einer verzögert auswertenden Programmiersprache.

6.1.5 Striktheitsanalyse

Beim Thema „Striktheit" bzw. „Striktheitsanalyse" sei noch erwähnt, daß die Gleichheitsanalyse, wie in [SS96a, S. 28] angesprochen, auch dazu benutzt werden kann, Striktheitsinformationen zu gewinnen. Durch eine Anfrage der Art

$$\texttt{f}\ x_1 \ldots x_{i-1}\ \texttt{bot}\ x_{i+1} \ldots x_n \ \doteq\ \texttt{bot}$$

wird nach Striktheit im i-ten Argument gesucht. Die durch die zusätzlichen Kalkülregeln gegenüber [Sch94] erweiterten Möglichkeiten der Analyse werden dabei kaum genutzt, da die eine Seite der Anfrage stets `bot` ist. So wurden einige der dortigen Beispiele getestet und verhielten sich im Großen und Ganzen auch genauso. Zum Teil wurden aber qualifiziertere Aussagen geliefert, z.B. über die Striktheit von `if` im zweiten oder dritten Argument. Denn hier erhalten wir die Lösung mit `True` bzw. `False` als erstem Argument.

6.2 Quantitative Leistungsfähigkeit

Die Implementierung wurde mit dem Chalmers Haskell B Compiler der Version 0.9999.4 auf verschiedenen Plattformen übersetzt. Dazu zählen sowohl PC's mit einem Prozessor von AMD oder Intel unter dem Betriebssystem Linux, als auch HP Workstations unter dem Betriebssystem HP-UX. Nach

Prozessor bzw. Rechnertyp	Version des Betriebssystems	RAM	Optimierung keine	Optimierung voll
AMD X5/133	Linux 2.0.32	48 MB	109 s	57 s
AMD K6-2/333	Linux 2.0.35	128 MB	9 s	6 s
HP A 9000/715	HP-UX B.10.20	64 MB	154 s	46 s
HP A 9000/780	HP-UX B.10.20	384 MB	23 s	8 s
Intel Pentium 100	Linux 2.0.35	64 MB	29 s	23 s
Intel Pentium II 233	Linux 2.0.35	128 MB	10 s	n.e. [2]

Tabelle 6.1: Die Laufzeiten auf verschiedenen Systemen

Möglichkeit wurde jeweils ein nicht optimiertes und ein voll optimiertes Kompilat getestet. Mit diesem wurden die 94 Formeln analysiert, die zusammen mit den jeweiligen Analyseergebnissen in Anhang A zu finden sind. Die Laufzeiten hierfür kann man der Tabelle 6.1 entnehmen. Aus den Zahlen ist zu ersehen, daß die Analyse auf einem zur Zeit handelsüblichen[3] PC im Durchschnitt weniger als 65 *ms* für eine Formel benötigt.

Bei der Bewertung der Laufzeiten auf den Workstations ist zu bedenken, daß diese zum einen die Formeln und das Kernsprachenprogramm über das Netzwerk von einem Fileserver einlesen mußten und zum anderen gleichzeitig auch noch anderen Aufgaben dienten.

[2] „n.e.“ – nicht ermittelt

[3] Damit ist der mit dem AMD K6 bestückte Rechner gemeint. Dessen Hauptspeichergröße mag dabei noch nicht handelsüblich sein; aber die Gleichheitsanalyse ist so speicherinternsiv nicht. Dagegen sind die meisten der heute zum Verkauf angebotenen PC's mittlerweile schon mit höher getakteten Prozessoren ausgestattet.

Kapitel 7

Zusammenfassung und Ausblick

Ziel der Arbeit war es, einen Gleichheitskalkül für eine nicht-strikte funktionale Programmiersprache zu entwerfen und praktisch umzusetzen. Dazu wurden in Kapitel 2 die theoretischen Grundlagen, d.h. im wesentlichen der λ-Kalkül, vorgestellt. In Kapitel 3 folgte dann eine knappe Einführung in die funktionale Programmierung, da die Arbeit ja in diesem Bereich angesiedelt ist.

Damit waren die Voraussetzungen geschaffen, um in Kapitel 4 die funktionale Kernsprache anzugeben, auf der die Analyse stattfindet. Von besonderer Wichtigkeit ist dabei Abschnitt 4.4, in dem die Gleichheit auf Ausdrücken dieser Sprache erklärt wird.

Kapitel 5 schließlich besteht im wesentlichen aus zwei großen Teilen. Da sind zum einen die Abschnitte 5.1, 5.2 und 5.3, die sich mit dem Gleichheitskalkül an sich beschäftigen. Hier wird das Prinzip von Tableaus erläutert und die Korrektheit der Kalkülregeln sowie die Menge der Lösungen behandelt. Die Verwendung von Prinzipien und Ergebnissen aus der Kontextanalyse ermöglicht es, präzise Beschreibungen der Lösungen auch dann anzugeben, wenn die Gleichheit nicht in allen Fällen gilt.

In den übrigen Abschnitten 5.4, 5.5 und 5.6 geht es um die Planung und Durchführung der Implementierung. Mit dieser ist es nun auf einfache Weise möglich, die Gleichheit von zwei Ausdrücken der Kernsprache automatisch beweisen zu lassen.

Die so erzielten Ergebnisse wurden abschließend in Kapitel 6 zusammengetragen. An ihnen zeigt sich, daß mit der gewählten Methode auch zu schwierigen, nicht-trivialen Aufgaben aussagekräftige Lösungen gefunden werden. Nichtdestotrotz haben wir auch einfache Beispiele gesehen, die nicht bewiesen werden können, weil der Kalkül nicht über Induktionsprinzipien verfügt.

7.1 Ausblick

Damit sind wir auch schon beim Ausblick und möglichen Erweiterungen angelangt.

7.1.1 Induktion

Denn eine denkbare Erweiterung wäre die Kombination des Gleichheitskalküls mit einem Induktionsbeweiser. Mit diesem könnten dann Aussagen bewiesen werdem, bei denen der Kalkül im Umgang mit endlichen Datenstrukturen scheitert. Auf der anderen Seite stellt es eine gewisse Schwierigkeit dar, diesen Prozeß zu automatisieren. Es wird nicht einfach zu ermitteln sein, welche Aussagen am besten auf welche Art bewiesen werden können.

7.1.2 Anforderungen in der Wurzel

Manchmal könnte es eine Hilfe sein, Anforderungen an freie Variable der Wurzel zuzulassen. Denn auf diese Weise könnten die Fälle, für die Gleichheit bewiesen werden soll, im vorhinein eingeschränkt werden. Beispielsweise könnte für die Anfrage $\texttt{append}\ xs\ ys\ \doteq\ xs$ die zusätzliche Anforderung $xs \in INF = \langle Bot, \texttt{Kons}\ Top\ INF\rangle$ gestellt werden. Die Lösung wäre dann nur noch von ys abhängig, und in diesem Fall könnte die Analyse einfach mit „Ja“ antworten.

7.1.3 Zusammenspiel mit der Kontextanalyse

Bei obigen Beispiel könnte die Motivation darin liegen, daß die Information $xs \in INF$ bereits vorliegt. In Verbindung mit der Kontextanalyse liegt aber vielleicht eher der entgegengesetzte Weg auf der Hand. Denn eine Lösung der Gleichheitsanalyse kann wiederum als Eingabe für die Kontextanalyse verwendet werden. Dies wäre natürlich für die Anforderungen, die im Zuge von Approximationen entstehen, wie z.B. bei **cpe8** auf Seite 117, von besonderer Bedeutung.

7.1.4 Nichtdeterministische Reduktion

Weiterhin kann es von Interesse sein, eine nichtdeterministische Reduktionsrelation zu betrachten. Damit könnten z.B. zu den Untersuchungen in [SS96b] auch Gleichheitsbetrachtungen angestellt werden.

In der Implementierung bedürfte es dazu aber einiger Änderungen. Die einfachste davon ist noch, ein Primitiv wie `choice` zu realisieren[1]. Schwieri-

[1] Dies reduziert nichtdeterministisch zu einem seiner beiden Argumente. Bei einer Implementierung in der Gleichheitsanalyse müßten aber alle Möglichkeiten berücksichtigt werden. Daher würde dieses Primitiv hier so implementiert, daß es *eine Menge* liefert, die aus beiden Argumenten besteht.

ger wird es bei der Handhabung dieser nichtdeterministischen Reduktionsergebnisse. Als Beispiel seien hier die Ausdrücke `choice 1 2` und `choice 2 1` genannt, die beide zu der Menge $\{1, 2\}$ reduzieren, also äquivalent sind.

Welche Möglichkeiten sind nun denkbar, im Rahmen der gebebenen Implementierung Mengen von Ausdrücken als Ergebnis einer nichtdeterministischen Reduktion darzustellen?

- Eine Idee ist, Listen von Ausdrücken zu bilden, und diese zu einer weiteren Instanz der Klasse `CoreExpression` zu machen. Dabei wird man aber schnell auf technische Schwierigkeiten stoßen. Zum einen dürfte nicht sofort klar sein, was `isWHNF` auf einer Liste von Ausdrücken bedeuten soll. Zum anderen kann nur der gesamten Liste eine einzige, aber nicht jedem einzelnen Ausdruck darin eine eigene Umgebung zugeordnet werden.

- Ein anderer Ansatz könnte sein, den Typ `Various` um eine Menge zu erweitern. In der bestehenden Form kann er zwar auch eine Liste aufnehmen, wie `VOr` bzw. `VAnd` interpretiert werden soll, ist aber bereits durch die Implementierung festgelegt.

7.1.5 Verallgemeinerung

In dieser Arbeit wurde die Gleichheitsanalyse unter Verwendung der Kontextanalyse entwickelt. Man kann sie aber auch als Beispiel dafür auffassen, wie mit der Methode der Kontextanalyse ein Prädikat über Ausdrücken der Kernsprache gelöst werden kann. Selbstverständlich ist dies nicht für jedes beliebige Prädikat möglich. Aber es ist durchaus denkbar, daß z.B. auch die Terminierungsanalyse aus [Kic96] in diesem Rahmen durchgeführt werden könnte.

In diesem Zusammenhang ergeben sich dann recht komplexe Fragestellungen, wie z.B. welcher Art die Prädikate sein, oder welche Regeln gegenüber dem Kontextkalkül hinzugefügt werden dürfen. In diesem Bereich könnten weiterführende Untersuchungen sinnvoll sein.

Anhang A

Beispielanalysen

In diesem Anhang sind eine Reihe von Analysebeispielen in Form eines vollständigen Protokolls zu einem Testdurchlauf abgedruckt. Dabei erfolgte der Aufruf des Analyseprogrammes mittels

```
ape samples/batch.core samples/formulas
```

Die Ausgabe des Programmes wurde übersichtlich formatiert aber ansonsten unverändert übernommen. Zur Interpretation der Ergebnisse seien noch einige Dinge angemerkt.

Zum einen ersetzt das Programm der Einfachheit halber keine Variablen in einem `where`-Ausdruck durch `Top`, auch wenn diese nur einmal auftauchen und nicht durch das Anforderungs-Muster gebunden sind. Zum anderen führten einige kleinere Korrekturen an dem Programm in der Zwischenzeit dazu, daß bessere Laufzeiten erreicht wurden. Die Tests aus Abschnitt 6.2 konnten aber nicht noch einmal durchgeführt werden, weshalb am Ende nur der verbesserte Wert für den AMD X5/133 erscheint. Außerdem ist erwähnenswert, daß z.B. für die Anfrage

$$\texttt{compose (foldr f1 z) (map f2)} \doteq \texttt{foldr (compose f1 f2) z}$$

keine Lösung gefunden wird, wohl aber für

$$\texttt{foldr f1 z (map f2 l)} \doteq \texttt{foldr (compose f1 f2) z l}$$

da die Gleichheit nur mit Einschränkungen an das Argument `l` gilt.

```
Reading samples/batch.core ... done.
Reading samples/formulas ... done.

("Y =? F", [], Yes)

("id =? K1 x1", ["x1"], (Top))

("x =? (case Cons{1,0} of <1> -> K; <2> x xs -> K) x y", ["x", "y"],
```

```
   (Top, Top))

("id =? (case Cons{1,0} of <1> -> K1; <2> x xs -> K1) x", ["x"],
   (Top))

("id =? (case x of <1> -> K1; <2> x xs -> K1) y", ["x", "y"],
   < (Cons{1,0}, Top), (Cons{2,2} Top Top, Top) >)

("case x of <1> -> Cons{1,0}; <2> x xs -> Cons{2,2} x xs =? id x", ["x"],
   < (Bot), (Cons{1,0}), (Cons{2,2} Top Top) >)

("add x Zero =? x", ["x"],
   < (Bot), (Cons{5,0}), (Cons{6,1} x1) where ([x1], add x Zero =? x) >)

("add x (Succ y) =? Succ (add x y)", ["x", "y"],
   < (Cons{5,0}, Top), (Cons{6,1} x1, y) where
     ([x1, y], add x (Succ y) =? Succ (add x y)) >)

("add x (add y z) =? add (add x y) z", ["x", "y", "z"],
   < (Bot, Top, Top), (Cons{5,0}, Top, Top), (Cons{6,1} x1, y, z) where
     ([x1, y, z], add x (add y z) =? add (add x y) z) >)

("map Succ x1 =? map (add One) x1", ["x1"],
   < (Bot), (Cons{1,0}), (Cons{2,2} x2 x3) where
     ([x3], map Succ x1 =? map (add One) x1) >)

("idOnCons =? id", [], No)

("idOnCons x =? id x", ["x"],
   < (Bot), (Cons{1,0}), (Cons{2,2} Top Top) >)

("idOnList =? id", [], No)

("idOnList x =? id x", ["x"],
   < (Bot), (Cons{1,0}), (Cons{2,2} x1 x2) where
     ([x2], idOnList x =? id x) >)

("append xs (append ys zs) =? append (append xs ys) zs", ["xs", "ys", "zs"],
   < (Bot, Top, Top), (Cons{1,0}, Top, Top),
     (Cons{2,2} x1 x2, ys, zs) where
     ([x2, ys, zs], append xs (append ys zs) =? append (append xs ys) zs) >)

("append xs Nil =? xs", ["xs"],
   < (Bot), (Cons{1,0}), (Cons{2,2} x1 x2) where
     ([x2], append xs Nil =? xs) >)

("append xs ys =? xs", ["xs", "ys"],
   < (Bot, Top), (Cons{1,0}, Cons{1,0}), (Cons{2,2} x1 x2, ys) where
     ([x2, ys], append xs ys =? xs) >)

("append xs ys =? zs", ["xs", "ys", "zs"],
   < (Bot, Top, Bot), (Cons{1,0}, zs, zs), (Cons{1,0}, ys, ys) >)

("length (append xs ys) =? add (length xs) (length ys)", ["xs", "ys"],
```

```
  < (Bot, Top), (Cons{1,0}, Top), (Cons{2,2} Top Bot, Top) >)

("lazyLength (append xs ys) =? add (lazyLength xs) (lazyLength ys)",
  ["xs", "ys"],
  < (Bot, Top), (Cons{1,0}, Top), (Cons{2,2} x1 x2, ys) where
    ([x2, ys],
     lazyLength (append xs ys) =? add (lazyLength xs) (lazyLength ys)) >)

("compose (filter p) (map f) =? compose (map f) (filter (compose p f))",
  ["p", "f"],
  No)

("filter p (map f l) =? map f (filter (compose p f) l)", ["p", "f", "l"],
  < (Top, Top, Bot), (Top, Top, Cons{1,0}),
    (p, f, Cons{2,2} x1 x2) needs [(p (f x1) , Bot)],
    (p, f, Cons{2,2} x1 x2) where
      ([p, f, x2], filter p (map f l) =? map f (filter (compose p f) l))
      needs [(p (f x1) , Cons{3,0})],
    (p, f, Cons{2,2} x1 x2) where
      ([p, f, x2], filter p (map f l) =? map f (filter (compose p f) l))
      needs [(p (f x1) , Cons{4,0})] >)

("compose (foldr f1 z) (map f2) =? foldr (compose f1 f2) z",
  ["f1", "z", "f2"],
  No)

("foldr f1 z (map f2 l) =? foldr (compose f1 f2) z l",
  ["f1", "z", "f2", "l"],
  < (Top, Top, Top, Bot), (Top, Top, Top, Cons{1,0}),
    (f1, z, f2, Cons{2,2} x1 x2) where
    ([f1, z, f2, x2], foldr f1 z (map f2 l) =? foldr (compose f1 f2) z l) >)

("compose (map f) (map g) =? map (compose f g)", ["f", "g"], No)

("map f (map g l) =? map (compose f g) l", ["f", "g", "l"],
  < (Top, Top, Bot), (Top, Top, Cons{1,0}), (f, g, Cons{2,2} x1 x2) where
    ([f, g, x2], map f (map g l) =? map (compose f g) l) >)

("add (sum x) (length x) =? sum (map Succ x)", ["x"],
  < (Bot), (Cons{1,0}) >)

("listOfNumEq xs xs =? True", ["xs"], < (Cons{1,0}) >)

("listOfNumEq xs ys =? listOfNumEq ys xs", ["xs", "ys"],
  < (Bot, Bot), (Bot, Cons{1,0}), (Bot, Cons{2,2} Top Top),
    (Cons{1,0}, Bot), (Cons{1,0}, Cons{1,0}),
    (Cons{1,0}, Cons{2,2} Top Top), (Cons{2,2} Top Top, Bot),
    (Cons{2,2} Top Top, Cons{1,0}),
    (Cons{2,2} Cons{5,0} Top, Cons{2,2} Bot Top),
    (Cons{2,2} Cons{5,0} Top, Cons{2,2} (Cons{6,1} Top) Top) >)

("and (listOfNumEq xs ys) (listOfNumEq ys zs) =? listOfNumEq xs zs",
  ["xs", "ys", "zs"],
  < (Bot, Top, Top), (Cons{1,0}, Cons{1,0}, Top), (Cons{1,0}, Bot, Bot),
```

```
     (Cons{1,0}, Cons{2,2} Top Top, Cons{2,2} Top Top) >)

("isortNum =? qsortNum", [], No)

("isortNum xs =? qsortNum xs", ["xs"], < (Bot), (Cons{1,0}) >)

("repeat =? repeatY", [], Yes)

("repeat x =? repeatY x", ["x"], (Top))

("repeat =? repeat2", [], Yes)

("repeat x =? repeat2 x", ["x"], (Top))

("repeat =? repeat3", [], Yes)

("repeat x =? repeat3 x", ["x"], (Top))

("repeatY =? repeat2", [], Yes)

("repeatY x =? repeat2 x", ["x"], (Top))

("Y repeat =? Y repeatY", [], Yes)

("Y repeat =? Y repeat2", [], Yes)

("and x y =? and y x", ["x", "y"],
   < (Bot, Bot), (Bot, Cons{3,0}),
     (Cons{3,0}, Bot), (Cons{3,0}, Cons{3,0}), (Cons{3,0}, Cons{4,0}),
     (Cons{4,0}, Cons{3,0}), (Cons{4,0}, Cons{4,0}) >)

("strictAnd x y =? strictAnd y x", ["x", "y"],
   < (Bot, Bot), (Bot, Cons{3,0}), (Bot, Cons{4,0}),
     (Cons{3,0}, Bot), (Cons{3,0}, Cons{3,0}), (Cons{3,0}, Cons{4,0}),
     (Cons{4,0}, Bot), (Cons{4,0}, Cons{3,0}), (Cons{4,0}, Cons{4,0}) >)

("and x (and y z) =? and (and x y) z", ["x", "y", "z"],
   < (Bot, Top, Top), (Cons{3,0}, Top, Top), (Cons{4,0}, Top, Top) >)

("strictAnd x (strictAnd y z) =? strictAnd (strictAnd x y) z",
   ["x", "y", "z"],
   < (Bot, Top, Top) >)

("or x y =? or y x", ["x", "y"],
   < (Bot, Bot), (Bot, Cons{4,0}),
     (Cons{3,0}, Cons{3,0}), (Cons{3,0}, Cons{4,0}),
     (Cons{4,0}, Bot), (Cons{4,0}, Cons{3,0}), (Cons{4,0}, Cons{4,0}) >)

("strictOr x y =? strictOr y x", ["x", "y"],
   < (Bot, Bot), (Bot, Cons{3,0}), (Bot, Cons{4,0}),
     (Cons{3,0}, Bot), (Cons{3,0}, Cons{3,0}), (Cons{3,0}, Cons{4,0}),
     (Cons{4,0}, Bot), (Cons{4,0}, Cons{3,0}), (Cons{4,0}, Cons{4,0}) >)

("or x (or y z) =? or (or x y) z", ["x", "y", "z"],
```

```
   < (Bot, Top, Top), (Cons{3,0}, Top, Top), (Cons{4,0}, Top, Top) >)

("strictOr x (strictOr y z) =? strictOr (strictOr x y) z", ["x", "y", "z"],
   < (Bot, Top, Top) >)

("f_1 Cons{4,0} =? g_1 Cons{4,0} Cons{4,0}", [], Yes)

("f_4 =? Cons{2,2}", [], Yes)

("map f l =? map g l", ["f", "l", "g"], < (g, Top, g), (f, Top, f) >)

("I bot =? bot", ["bot"], (Bot))

("K bot y =? bot", ["bot", "y"], (Bot, Top))

("K x bot =? bot", ["x", "bot"], (Bot, Bot))

("K1 bot x =? bot", ["bot", "x"], (Bot, Bot))

("K1 x bot =? bot", ["x", "bot"], (Top, Bot))

("S bot q r =? bot", ["bot", "q", "r"], No)

("S p bot r =? bot", ["p", "bot", "r"], No)

("S p q bot =? bot", ["p", "q", "bot"], No)

("compose bot g x =? bot", ["bot", "g", "x"], No)

("compose f bot x =? bot", ["f", "bot", "x"], No)

("compose f g bot =? bot", ["f", "g", "bot"], No)

("twice bot x =? bot", ["bot", "x"], No)

("twice f bot =? bot", ["f", "bot"], No)

("if bot then else =? bot", ["bot", "then", "else"], (Bot, Top, Top))

("if pred bot else =? bot", ["pred", "bot", "else"],
   < (Bot, Bot, Top), (Cons{3,0}, Bot, Top), (Cons{4,0}, Bot, Bot) >)

("if pred then bot =? bot", ["pred", "then", "bot"],
   < (Bot, Top, Bot), (Cons{3,0}, Bot, Bot), (Cons{4,0}, Top, Bot) >)

("not bot =? bot", ["bot"], (Bot))

("and bot y =? bot", ["bot", "y"], (Bot, Top))

("and x bot =? bot", ["x", "bot"], < (Bot, Bot), (Cons{3,0}, Bot) >)

("strictAnd bot y =? bot", ["bot", "y"], (Bot, Top))

("strictAnd x bot =? bot", ["x", "bot"],
```

```
   < (Bot, Bot), (Cons{3,0}, Bot), (Cons{4,0}, Bot) >)

("or bot y =? bot", ["bot", "y"], (Bot, Top))

("or x bot =? bot", ["x", "bot"], < (Bot, Bot), (Cons{4,0}, Bot) >)

("strictOr bot y =? bot", ["bot", "y"], (Bot, Top))

("strictOr x bot =? bot", ["x", "bot"],
   < (Bot, Bot), (Cons{3,0}, Bot), (Cons{4,0}, Bot) >)

("rev bot =? bot", ["bot"], (Bot))

("length bot =? bot", ["bot"], (Bot))

("head bot =? bot", ["bot"], (Bot))

("map bot l =? bot", ["bot", "l"], < (Bot, Bot) >)

("map f bot =? bot", ["f", "bot"], (Top, Bot))

("filter bot l =? bot", ["bot", "l"],
   < (Bot, Bot), (Bot, Cons{2,2} x1 x2) needs [(Bot x1, Bot)],
     (Bot, Cons{2,2} x1 x2) where
     ([Bot, x2], filter bot l =? bot) needs [(Bot x1, Cons{4,0})] >)

("filter p bot =? bot", ["p", "bot"], (Top, Bot))

("foldr bot z l =? bot", ["bot", "z", "l"],
   < (Bot, Top, Bot), (Bot, Bot, Cons{1,0}) >)

("foldr f bot l =? bot", ["f", "bot", "l"],
   < (Top, Bot, Bot), (Top, Bot, Cons{1,0}) >)

("foldr f z bot =? bot", ["f", "z", "bot"],
   (Top, Top, Bot))

("any bot l =? bot", ["bot", "l"], No)

("any p bot =? bot", ["p", "bot"], No)

("all bot l =? bot", ["bot", "l"], No)

("all p bot =? bot", ["p", "bot"], No)

("add bot y =? bot", ["bot", "y"], (Bot, Top))

("add x bot =? bot", ["x", "bot"], < (Bot, Bot), (Cons{5,0}, Bot) >)

The analysis took 0 hours, 0 minutes and 40 seconds for 94 formulas.
That gives an average of 0.425 sec/formula.
```

Anhang B

Hilfsmodule

In diesem Teil des Anhanges sind Module mit Hilfsfunktionen, die von allgemeinem Interesse sein können, zu finden.

B.1 Nützliche Kombinatoren

Das Modul `FP` stellt einige nützliche Kombinatoren bereit.

```
> module FP where

> infixr 3 .&&.
> infixr 2 .||.
```

Die Funktion `binaryCombine` kombiniert die Ergebnisse, die durch die Anwendung zweier verschiedener Prädikate auf ein Argument enstehen.

```
> binaryCombine binOp pred1 pred2 x = pred1 x `binOp` pred2 x
```

Damit lassen sich die logischen Kombinatoren `.||.` und `.&&.` nun einfach ausdrücken.

```
> (.||.) = binaryCombine (||)
> (.&&.) = binaryCombine (&&)
```

Diese kombinieren, wie schon erwähnt, zwei einstellige Funktionen mit booleschem Rückgabewert. Die Verwendung des Punktes `.` im Namen soll dabei an die Funktionskomposition erinnern.

```
> pairing (f,g) x = (f x, g x)
> paired  f (x,y) = (f x, f y)
```

`pairing` und `paired` dienen dazu, ein Paar von zwei Funktionen auf ein Argument anzuwenden bzw. eine Funktion auf ein Paar. Die Funktion `mapBinary` schließlich dient dazu, eine zweistellige Funktion auf zwei Listen anzuwenden.

```
> mapBinary f l1 l2 = map (uncurry f) (zip l1 l2)
```

B.2 Substitutionen

In diesem Modul werden Substitutionen definiert.

```
> module Substitution where
> import List
```

Diese sind Listen von Paaren, die die einzelnen Ersetzungen darstellen.

```
> data SubstPair n e = n :->: e
>                      deriving Show

> type Substitution n e = [SubstPair n e]
```

Bei der leeren Substitution wird gar nichts ersetzt, daher ist die Liste leer.

```
> emptySubst :: Substitution n e
> emptySubst = []
```

Die Funktion `inconsistent` überprüft, ob eine Substitution dahingehend inkonsistent ist, daß ein Element in zwei Paaren unterschiedlich ersetzt würde.

```
> inconsistent s s' = (not . null)
>   [ x | (x  :->: y)  <- s, (x' :->: y') <- s',
>         x == x' && y /= y' ]
```

Die Funktion `unionSubst` benötigt dies nämlich, um die Vereinigung zweier Substitutionen korrekt berechnen zu können.

```
> unionSubst s s'
>   | inconsistent s s' = Nothing
>   | otherwise         = Just (union s s')

> unionMaybeSubst Nothing _ = Nothing
> unionMaybeSubst _ Nothing = Nothing
> unionMaybeSubst (Just s) (Just s') =
>   unionSubst s s'
```

Definitions- und Wertebereich einer Substitution sind einfach anzugeben.

```
> domainSubst :: Substitution n e -> [n]
> domainSubst = map (\(x :->: _) -> x)

> rangeSubst  :: Substitution n e -> [e]
> rangeSubst  = map (\(_ :->: x) -> x)
```

B.3 Die Zustandsmonade

In diesem Modul wird eine Zustandsmonade definiert, wie sie u.a. in [Jon93, Abschnitt 2.1] beschrieben ist.

```
> module StateMonad where
```

Dafür wird der Typ `StateTransformer` erklärt, der einfach eine Funktion kapselt, die bei Anwendung auf einen Zustand vom Typ `s` ein Paar aus einem Ergebnis vom Typ `a` und einem neuen Zustand liefert.

```
> data StateTransformer s a = ST (s -> (a, s))

> instance Functor (StateTransformer s) where
>   map f (ST stFunc) = ST (\s -> let (x, s') = stFunc s
>                                 in (f x, s'))
```

Das bedeutet, `map` gibt, angewandt auf eine Funktion `f` und einen `StateTransformer`, einen neuen `StateTransformer` zurück, der den Zustandsübergang durchführt und `f` auf das Ergebnis anwendet. Die Monadeninstanz für `StateTransformer` wird nun entsprechend zu [Jon93] angegeben.

```
> instance Monad (StateTransformer s) where
>   (ST stFunc) >>= k = ST (\s -> let (x, s') = stFunc s
>                                     ST stFunc' = k x
>                                 in stFunc' s')
>   return x = ST (\s -> (x, s))
```

Um am Ende einer Kette von Zustandsübergängen an das Ergebnis zu gelangen, benutzt man `startingWith`.

```
> startingWith :: (StateTransformer s a) -> s -> a
> (ST stFunc) `startingWith` s = fst (stFunc s)
```

An manchen Stellen ist es wünschenswert, auch auf den letzten Zustand einer solchen Kette zugreifen zu können. Daher wird hier zusätzlich `beginningWith` definiert.

```
> beginningWith :: (StateTransformer s a) -> s -> (a, s)
> (ST stFunc) `beginningWith` s = (stFunc s)
```

Literaturverzeichnis

[Abr90] ABRAMSKY, SAMSON: *The lazy lambda calculus.* In: TURNER, DAVID A. (Herausgeber): *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, Kapitel 4, Seiten 65–116. Addison-Wesley, 1990.

[AS85] ABELSON, HAROLD und GERALD JAY SUSSMAN: *Structure and Interpretation of Computer Programs.* McGraw-Hill, New York, 1985.

[AU72] AHO, ALFRED V. und JEFFREY D. ULLMAN: *The Theory Of Parsing, Translating And Compiling*, Band 1. Prentice Hall, Englewood Cliffs, NJ, 1972.

[Bar84] BARENDREGT, HENDRIK PIETER: *The Lambda Calculus, Its Syntax and Semantics.* Elsevier Science Publishers, 1984.

[Buc95] BUCHINGER, THORSTEN: *Abstrakte Maschinen zur Auswertung nicht-strikter funktionaler Sprachen: Analyse, Vergleich und Optimierung.* Diplomarbeit, Johann Wolfgang Goethe-Universität Frankfurt, 1995.

[BW88] BIRD, RICHARD und PHILIP WADLER: *Introduction to Functional Programming.* Prentice-Hall International, London, 1988.

[Dav89] DAVIS, RUTH E.: *Truth, Deduction, And Computation.* Computer Science Press, 1989.

[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.

[FH88] FIELD, A.J. und P.G. HARRISON: *Functional Programming.* Addison Wesley, 1988.

[GM96] GILL, ANDY und SIMON MARLOW: *Happy Manual.* Happy version 0.9, 1996.

[Gor94] GORDON, ANDREW D.: *A Tutorial on Co-Induction and Functional Programming.* In: *Functional Programming, Glasgow 1994*, Workshops in Computing, Seiten 78–95. Springer, 1994.

[Hin69] HINDLEY, R.: *The principal type scheme of an object in combinatory logic.* Transactions of the American Mathematical Society, 146:29–60, Dezember 1969.

[HU79] HOPCROFT, JOHN E. und JEFFREY D. ULLMAN: *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, Massachusetts, 1979.

[JJM97] JONES, SIMON PEYTON, MARK JONES und ERIK MEIJER: *Type classes: An Exploration of the Design Space.* Technischer Bericht University of Glasgow, University of Nottingham, University of Utrecht, Oregon Graduate Institute, 1997.

[Jon93] JONES, MARK P.: *A System of Constructor Classes: Overloading and Implicit Higher-order Polymorphism.* Technischer Bericht Yale University, Department of Computer Science, 1993.

[Kic96] KICK, HUBERT: *Terminierungsanalyse für den Sprachkern einer nicht-strikten Programmiersprache unter Verwendung eines Tableaukalküls für abstrakte Reduktion.* Diplomarbeit, Johann Wolfgang Goethe-Universität Frankfurt, 1996.

[Klo97] KLOSE, NORBERT: *Lucky Manual.* Lucky Version 0.3, 1997.

[MTH90] MILNER, ROBIN, MADS TOFTE und ROBERT HARPER: *The Definition of Standard ML.* IT Press, Cambridge, Massachusetts, 1990.

[Pan96] PANITZ, SVEN ERIC: *Termination proofs for a lazy functional language by abstract reduction.* Technischer Bericht Johann Wolfgang Goethe-Universität Frankfurt, 1996.

[PHA+97] PETERSON [ED.], JOHN, KEVIN HAMMOND [ED.], LENNART AUGUSTSSON, BRIAN BOUTEL, WARREN BURTON, JOSEPH FASEL, ANDREW D. GORDON, JOHN HUGHES, PAUL HUDAK, THOMAS JOHNSSON, MARK JONES, ERIK MEIJER, SIMON PEYTON JONES, ALASTAIR REID und PHILIP WADLER: *Report on the Programming Language Haskell. A Non-strict, Purely Functional Language. Version 1.4*, April 1997.

[PJ87] PEYTON JONES, SIMON L.: *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[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.

[Sch] Schütz, Marko: *Kontext-Analyse funktionaler Sprachen.* Notizen im Vorfeld der Dissertation, 1998.

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

[Sch95] Schäfer, Steffen: *Objektorientierte Entwurfsmethoden.* Addison-Wesley, 1995.

[SS96a] Schmidt-Schauss, Manfred: *CPE: A Calculus for Proving Equivalence of Expressions in a Non-Strict Functional Language.* Technischer Bericht Johann Wolfgang Goethe-Universität Frankfurt, 1996.

[SS96b] Schmidt-Schauss, Manfred: *A Partial Rehabilitation of Side-Effecting I/O: Non-Determinism in Non-Strict Functional Languages.* Technischer Bericht Johann Wolfgang Goethe-Universität Frankfurt, 1996.

[SS96] Schmidt-Schauss, Manfred: *Einführung in das Funktionale Programmieren.* Johann Wolfgang Goethe-Universität Frankfurt, Vorlesung, WS 1995/96.

[SS97] Schmidt-Schauss, Manfred: *Funktionale Programmierung II: Semantik, statische Analyse, Verifikation und abstrakte Interpretation.* Johann Wolfgang Goethe-Universität Frankfurt, Vorlesung, WS 1996/97.

[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.

[SSS97] Schütz, Marko und Manfred Schmidt-Schauss: *Automatic Extraction of Context Information from Programs Using Abstract Reduction.* Technischer Bericht Johann Wolfgang Goethe-Universität Frankfurt, 1997.

[Tur85] TURNER, D. A.: *Miranda: A non-strict functional language with polymorphic types.* In: *Functional Programming Languages and Computer Architecture*, Nummer 201 in *Lecture Notes in Computer Science*, Seiten 1–16. Springer, 1985.