Jonah: Ein System zur Validierung von Reduktionsdiagrammen in nichtdeterministischen Lambda-Kalkülen mit Let-Ausdrücken, Letrec-Ausdrücken und Konstruktoren

Michael Huber

Diplomarbeit

1. August 2000

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

Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20)

Danksagung

Ich möchte mich bei allen Personen bedanken, die mich bei der Anfertigung dieser Arbeit unterstützt haben. Dabei gilt mein besonderer Dank
Prof. Dr. Manfred Schmidt-Schauß, Dr. Arne Kutzner, Marko Schütz und Matthias Mann für ihre Unterstützung und ihre wertvollen Anregungen.
Bei meiner Frau Nina möchte ich mich für ihre unendliche Geduld, ihre aufmunternden Worte und das aufmerksame Korrekturlesen bedanken.
Bei meinem Sohn Jonah möchte ich mich dafür entschuldigen, daß zu wenig Zeit für ihn blieb.

Michael Huber

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

Frankfurt am Main, den 23. August 2000

Michael Huber

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Entwicklungszyklen heutiger Software werden nicht nur wegen der immens anwachsenden Nachfrage immer kürzer. Immer neue Hardware verlangt nach Software, die diese auch bis ins letzte ausnutzt. Dabei ist es von zentraler Bedeutung, daß beim Entwickeln von Software sehr auf *Zuverlässigkeit*, *Wartbarkeit* und *Erweiterbarkeit* geachtet werden muß — auch wenn es scheint, daß heutige Software gerade das Gegenteil dieser Eigenschaften erfüllt. Daß schon kleine Software-Fehler große, nicht nur finanzielle, Katastrophen auslösen können, zeigt nicht nur die gesamte Jahr-2000-Problematik, die die Computer-Industrie und natürlich die Nutzer ihrer Software-Produkte am Ende dieses Jahrhunderts sehr beschäftigte.

Die meisten dieser Probleme werden durch Typ-Konvertierungen und Fehlinterpretation von Datentypen verursacht. Entwicklern von Software sollten also Werkzeuge und Methoden (z. B. [Bab87]) zur Verfügung gestellt werden, die solche Fehler erkennen, oder um zumindest den Programmierer beispielsweise vor Seiteneffekten zu warnen — oder diese erst gar nicht zu ermöglichen.

Zur automatischen Programmverifikation eignen sich funktionale Programmiersprachen besonders gut, da sie auf einer mathematisch soliden und intensiv untersuchten Grundlage aufbauen, den λ-Kalkülen ([Bar84]). Erweiterungen des λ-Kalküls sind intensiv untersucht worden ([Abr90], [MOW98], [AFM$^+$95] u.a.). Bei der Untersuchung bzw. Entwicklung neuer Kalküle werden als gängige Methode in Korrektheitsbeweisen Reduktionsdiagramme benutzt ([Kut00],[SSH00]). Daß diese Diagramme auch für alle wohldefinierten Terme des Kalküls gültig sind, muß wiederum gezeigt werden, was bei komplexen Kalkül-Definitionen sehr aufwendig sein kann. Es zeigt sich jedoch, daß es in vielen Fällen schon ausreicht, fehlende Reduktionsdiagramme frühzeitig zu erkennen, und die Reduktionsdiagramme oder die Definitionen der Kalküle anhand der Gegenbeispiele zu ändern.

In der Dissertation von Arne Kutzner ([Kut00, Kapitel 7]) ist ein Programm beschrieben, mit Hilfe dessen Reduktionsdiagramme für den darin vorgestellten nichtdeterministischen Kalkül Λ_{let} maschinell überprüft werden. Wir greifen diese Idee auf und entwickeln ein

System, das nichtdeterministische λ-Kalküle mit Konstruktoren, einem `case`-Ausdruck sowie einem rekursiven `let`-Ausdruck (`letrec`) implementiert. Wir definieren weiterhin eine Sprache für Reduktionsdiagramme, so daß die zu analysierenden Diagrammsätze als Dateien ausgelagert werden können.

Das in dieser Arbeit vorgestellte System JONAH, unterstützt die Entwicklung von λ-Kalkülen, indem Reduktionsdiagramme für eine große Menge von möglichen Termen der Kalkül-Sprache validiert werden. Es zeigt sich, daß schon bei kleinen Termstrukturen, die Möglichkeiten so vielfältig sein können, so daß die Gefahr besteht, wichtige Fälle zu übersehen. JONAH generiert Terme in allen Varianten und überprüft, ob die Terme durch Reduktionsdiagramme schließbar sind. Dem Entwickler eines Kalküls gibt es dadurch die Möglichkeit, die Reduktionsdiagramme oder die Kalkül-Definition entsprechend abzuändern, wenn Reduktionsdiagramme nicht jeden Fall abdecken. Zusätzlich ist es möglich, Ausdrücke des Kalküls schrittweise zu reduzieren und dadurch die Auswertung in Normalordnung interaktiv zu verfolgen, wie es schon im Gofer-System [Jon94], dem Vorgänger des Haskell-Interpreters *Hugs* [Jon00], möglich war.

1.2 Überblick

In *Kapitel 2* werden wir einen kurzen Überblick über funktionale Programmiersprachen und ihre theoretische Grundlage — den λ-Kalkül — geben.

Kapitel 3 stellt die Kalküle λ_{nd} und $\lambda_{nd,rec}$ und deren entsprechende Reduktionsregeln vor. Die Implementierung der Kernsprache, die Markierung eines NO-Redex sowie interner Redexe und der Reduktionsmechanismus werden hier ausführlich behandelt.

In *Kapitel 4* werden die kontextuelle Äquivalenz und Reduktionsdiagramme vorgestellt. Reduktionsdiagramme bilden ein Werkzeug zum Beweisen der Korrektheit von Programmtransformationen.

Zum Überprüfen der Reduktionsdiagramme sollen so viele Terme wie möglich betrachtet werden, so daß sehr viele verschiedene Termstrukturen in den Test einfließen können. Dazu benötigen wir einen Termgenerator, der eine umfangreiche Menge von Termen erzeugt. Dieser wird in *Kapitel 5* beschrieben.

Den Kern dieser Arbeit bildet *Kapitel 6*, in dem zuerst die Syntax für Reduktionsdiagramme in der Diagramm-Datei beschrieben wird. Weiterhin wird die Behandlung verallgemeinerter Diagramme gezeigt, wodurch die Diagrammsätze nicht so umfangreich ausfallen. Als letztes wird die Validierung eines einzelnen Reduktionsdiagramms beschrieben, und dann das Verfahren zur Validierung eines ganzen Diagrammsatzes vorgestellt.

In *Kapitel 7* sollen die Reduktionsdiagramme für gängige Programmtransformationen durch JONAH validiert werden. Die dabei aufgetretenen Mißerfolge — im Bezug auf die fehlgeschlagene Validierung — zeigen, daß das System bei der Entwicklung eines verzögert auswertenden, nichtdeterministischen λ-Kalküls einen sehr guten Dienst leisten kann.

Kapitel 8 gibt noch einmal einen Überblick auf das System JONAH und bietet einen Ausblick auf Möglichkeiten zur Erweiterung des Werkzeugs.

Im *Anhang* befinden sich Auszüge der wichtigsten Module des Programms, der Standard-Dateien *prelude.core* und *rules.rule*, sowie je eine Kurzanleitung zur Bedienung und Erweiterung von JONAH.

Kapitel 2

Grundlagen

Als Grundlage für funktionale Programmiersprachen dient die Theorie des λ-Kalküls. Wir führen kurz in die Grundlagen des λ-Kalküls ein und beschreiben den in [MOW98] beschriebenen λ_{need}-Kalkül, der mit dem Kalkül Λ_{let} aus [Kut00] die Ausgangsposition für unsere beiden Kalküle λ_{nd} und $\lambda_{nd,rec}$ bildet. Weiterhin werden wir funktionale Programmiersprachen vorstellen und ihre Vor- und Nachteile bei der Implementierung von Software-Projekten besprechen.

2.1 Der λ-Kalkül

Um 1930 wurde der (ungetypte) λ-Kalkül von Church als Grundlage der Logik und der Mathematik eingeführt. Wegen einiger Paradoxe wurde der Ansatz jedoch verworfen. Später jedoch wurde gezeigt, daß der λ-Kalkül für die theoretische Informatik sehr gut zu gebrauchen ist. Er ist nämlich äquivalent zur Turing-Berechenbarkeit. Also, ist ein Problem im λ-Kalkül definierbar[1], so kann eine Turing-Maschine angegeben werden, die das Problem löst [Bar84, Kapitel 1].

Bevor wir den eigentlichen λ-Kalkül beschreiben, benötigen wir noch einige Definitionen und Begriffe.

2.1.1 Definitionen und Begriffe

Der λ-Kalkül ist auf sogenannte λ-Terme definiert, deren Gesamtheit die Sprache Λ bilden:

Definition 2.1.1 Die Sprache Λ ist die *Menge aller λ-Terme*, die über dem Alphabet $V \cup \{\lambda, ., (,)\}$ induktiv definiert sind:

(1) $x \in \Lambda$,

(2) $M \in \Lambda \Longrightarrow (\lambda x.M) \in \Lambda$,

[1] engl.: λ-definable

(3) $M, N \in \Lambda \Longrightarrow (MN) \in \Lambda$,

wobei V eine abzählbare Menge von Variablen ist. Ein Ausdruck der Form $(\lambda x.M)$ wird Abstraktion genannt.

Die Variablen eines λ-Terms können frei oder gebunden vorkommen, dazu die folgende induktive Definition:

Definition 2.1.2 Die Menge $\mathcal{FV}(M)$ der *freien Variablen des Terms M* ist die kleinste Menge mit den folgenden Eigenschaften:

$$\begin{array}{rcl} \mathcal{FV}(x) & = & \{x\}, \\ \mathcal{FV}(\lambda x.M) & = & \mathcal{FV}(M) \setminus \{x\}, \\ \mathcal{FV}(M\ N) & = & \mathcal{FV}(M) \cup \mathcal{FV}(N). \end{array}$$

Eine Variable $x \in M$ heißt *frei*, wenn $x \in \mathcal{FV}(M)$, ansonsten ist sie *gebunden*.

Im folgenden gehen wir davon aus, daß die *freien* Variablen eines λ-Terms von den *gebundenen* verschieden sind, da es sonst zu Namenskonflikten während der Reduktion eines Terms kommen kann.

Weiterhin benutzen wir folgende Notation:

(1) $M,\ N,\ \ldots$ seien beliebige λ-Terme,

(2) $x,\ y,\ z,\ \ldots$ seien beliebige Variablen,

(3) λ-Terme werden außen nicht geklammert,

(4) $\equiv$ bezeichne die syntaktische Äquivalenz,

(5) $\lambda x.x \equiv \lambda y.y$ (α-Konvertibilität),

(6) $\vec{x} = x_1 x_2 \ldots x_n$ und $\vec{N} = N_1 N_2 \ldots N_n$,

(7) $\lambda x_1 x_2 \ldots x_n.M \equiv \lambda \vec{x}.M \equiv \lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots))$, und

(7) $M N_1 N_2 \ldots N_n \equiv M\vec{N} \equiv (\ldots((MN_1)N_2)\ldots N_n)$ (Links-Assoziativität).

Aus Punkt (5) ergibt sich, daß zwei λ-Terme bis auf Umbenennung der gebundenen Variablen syntaktisch gleich sind.

Definition 2.1.3 Die Menge der *geschlossenen* λ-Terme ist

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

Ein *geschlossener* λ-Term heißt auch *Kombinator*.

Wir definieren einige spezielle *geschlossene* λ-Terme:

$$
\begin{array}{rcl}
I & \equiv & \lambda x.x \\
K & \equiv & \lambda xy.x \\
S & \equiv & \lambda xyz.xz(yz) \\
Y & \equiv & \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \\
\Omega & \equiv & (\lambda x.xx)(\lambda x.xx)
\end{array}
$$

Wir benötigen noch den Begriff des *Teilterms* bzw. *Sub-Terms*:

Definition 2.1.4 Die Menge $\mathcal{SUB}(M)$ aller *Teilterme von* M sei induktiv definiert:

$$
\begin{array}{rcl}
\mathcal{SUB}(x) & = & \{x\}, \\
\mathcal{SUB}(\lambda x.M) & = & \{\lambda x.M\} \cup \mathcal{SUB}(M), \\
\mathcal{SUB}(M\ N) & = & \{M\ N\} \cup \mathcal{SUB}(M) \cup \mathcal{SUB}(N).
\end{array}
$$

Wir schreiben für $M \in \mathcal{SUB}(N)$ auch $M \subset N$. Ein Teilterm kann dabei mehrmals innerhalb des Terms vorkommen.

Zur Auswertung von λ-Termen benötigen wir noch den Begriff der Substitution, die folgendermaßen definiert ist:

Definition 2.1.5 Für die *Substitution* aller Vorkommen einer Variable $x \in \mathcal{FV}(M)$ in einem Term $M \in \Lambda$ schreiben wir $M[N/x]$. Eine solche Substitution existiert und wird induktiv über die Struktur von λ-Termen definiert (siehe [Bar84, Seite 27]).

Um λ-Terme einfacher zu beschreiben definieren wir sogenannte Kontexte.

Definition 2.1.6 Ein *Kontext* C ist ein Term der induktiv definiert ist:

(1) $x \in V$ ist ein Kontext,

(2) $[\cdot]$ ist ein Kontext, und

(3) wenn C_1 und C_2 Kontexte sind, dann sind auch $C_1\ C_2$ und $\lambda x.C_1$ Kontexte.

Wenn C ein Kontext und $M \in \Lambda$ ist, dann bedeutet $C[M]$, daß M in alle Löcher[2] von C eingesetzt wird. Eine wichtige Eigenschaft von Kontexten ist, daß bei einer Substitution eine freie Variable in M zu einer gebundenen in $C[M]$ werden kann.

Beispielsweise deckt der Kontext $C[\lambda x.x]$ alle λ-Terme ab, die den Term $\lambda x.x$ enthalten.

Nun können wir uns mit der eigentlichen Theorie des λ-Kalküls befassen.

[2]engl.: hole

2.1.2 Reduktion des Kalküls λ

Der λ-Kalkül beschreibt eine Gleichheit zwischen zwei Termen $M, N \in \Lambda$. Durch bestimmte Axiome und Regeln kann man also *beweisen*, daß $M = N$ gilt. Als Notation wird dabei oft die Schreibweise $\lambda \vdash M = N$ benutzt, die besagt: „Unter der Theorie λ gilt: M ist äquivalent zu N“.

Definition 2.1.7 Der Kalkül λ besteht aus den Formeln

$$M = N$$

wobei für $M, N \in \Lambda$ folgende Regeln gelten:

(1) *Konversion*

- (β) $(\lambda x.M)N = M[N/x]$

(2) *Äquivalenz*

- (a) $M = N$
- (b) $M = N \Longrightarrow N = M$
- (c) $M = N, N = L \Longrightarrow M = L$

(3) *Ersetzbarkeit*

- (a) $M = N \Longrightarrow LM = LN$
- (b) $M = N \Longrightarrow ML = NL$
- (c) $M = N \Longrightarrow \lambda x.M = \lambda x.N$

Um nun die Gleichheit $M = N$ zweier λ-Terme zu beweisen, müssen wir versuchen mit den oben angegebenen Regeln den Term M in den Term N zu überführen. Dies kann man mit der sogenannten β-Reduktion.

Definition 2.1.8 Ein Term $R \equiv (\lambda x.M)N$ heißt *β-Redex*[3] und $R' \equiv M[N/x]$ das dazugehörende *Kontraktum*. Der Schritt vom Redex zum Kontraktum nennen wir *β-Kontraktion* und schreiben dafür $R \xrightarrow{\beta} R'$.

Ein Term kann jedoch mehrmals reduziert werden, so daß wir eine Folge dieser Reduktionen oder besser Reduktionsschritten als Reduktion bezeichnen.

Definition 2.1.9 Mehrerer *β-Kontraktionen* in Folge bezeichnen wir als *β-Reduktion*. Wir schreiben dafür $R \xrightarrow{\beta}_* R'$ und bezeichnen das Ergebnis einer Reduktion als *Redukt*.

Wenn ein Term nicht mehr weiter reduziert werden kann, also keine Ersetzung freier Variablen mehr möglich ist, dann nennen wir diesen Term eine *Normalform*. Wir sagen auch, daß der Term in Normalform ist.

[3] Abk. für *red*ucible *ex*pression, engl. für „reduzierbarer Ausdruck“

Definition 2.1.10 (NF) Ein Term $M \in \Lambda$ ist eine *Normalform*[4] oder *NF*, wenn er keinen β-Redex der Form $(\lambda x.M)N$ enthält.

Wenn man die Normalform eines Terms als seine Bedeutung (Semantik) betrachten würde, dann käme es zu Unstimmigkeiten in der Theorie des λ-Kalküls [Bar84, Seiten 39 – 40]. Daher wird ein anderes Kriterium herangezogen, daß alle Terme, die keine Kopfnormalform (HNF) besitzen, als bedeutungslos betrachtet.

Definition 2.1.11 (HNF) Ein Term $M \in \Lambda$ ist eine *Kopfnormalform*[5] oder *HNF*, wenn $M \equiv \lambda\vec{x}.y\vec{N}$ und y eine Variable ist.

Für verzögert auswertende funktionalen Programmiersprachen[6] wie Haskell oder Clean kann der klassische λ-Kalkül nicht direkt als Grundlage dienen. Deshalb haben sich viele Arbeiten damit beschäftigt, den klassischen λ-Kalkül derart zu erweitern, daß er einerseits in der Tradition der λ-Kalkül bleibt, andererseits aber die Grundlage für moderne Programmiersprachen bildet. Im folgenden Kapitel soll ein solcher Ansatz besprochen werden.

Zuletzt sollen noch Superkombinatoren definiert werden, die in funktionalen Programmiersprachen die Rolle der Funktionsdefinitionen übernehmen.

Definition 2.1.12 (Superkombinator) Ein λ-Ausdruck $t = \lambda x_1.x_2.\ldots.x_n.s$ mit $n \geq 0$ ist ein n-stelliger Superkombinator genau dann wenn:

1. s keine Abstraktion ist,
2. $\mathcal{FV}(t) = \emptyset$, d. h. ein Superkombinator hat keine freien Variablen, und
3. alle in s vorkommenden Abstraktionen ebenfalls Superkombinatoren sind.

2.2 Der verzögert auswertende Kalkül λ_{need}

In [MOW98] und [AFM+95] werden verzögert auswertende λ-Kalküle[7] beschrieben. Wir werden ersteren genauer betrachten und ihn in Kapitel 3 um verschiedene Sprachelemente erweitern, die ihn einen Schritt näher an funktionale Programmiersprache heranbringen werden.

2.2.1 Die Sprache Λ_{need}

In Definition 2.2.1 wird die Sprache Λ_{need} eingeführt, die eine durch einen `let`-Ausdruck erweiterte Version der klassischen Sprache Λ beschreibt. Diese Erweiterung modelliert das sogenannte *sharing*, das eine Eigenschaft von verzögert auswertenden Kalkülen ist. Durch *sharing* läßt sich die Korrespondenz zwischen Termen und Graphen zum Ausdruck bringen,

[4] engl. normal form
[5] engl. head normal form
[6] engl. auch *lazy functional languages*
[7] engl.: call-by-need lambda calculus

wobei die gebundene Variable des `let`-Ausdrucks eine Referenz zu einem Teilterm innerhalb des Term-Graphen darstellt.

Definition 2.2.1 Die *Sprache* Λ_{need} enthält Werte V und Terme M, N, die folgende Form haben können:

$$\begin{array}{lcl} V & ::= & x \mid \lambda x\,.\,M \\ M, N & ::= & V \mid M\ N \mid \texttt{let}\ x\ \texttt{=}\ M\ \texttt{in}\ N \end{array}$$

Einen Teil der Sprache Λ_{need}, *Variablen* und *Abstraktionen*, bezeichnen wir als *Werte*, um anzudeuten, daß nur diese bei einer kopierenden Reduktionsregel ($\xrightarrow{cp}$) kopiert werden dürfen. Dies ist die grundlegende Idee bei verzögert auswertenden Programmiersprachen, nämlich Teilterme an eine Variable zu binden ($\texttt{let}\ x\texttt{=}E\ \texttt{in}\ C[x]$) und in weiteren Teiltermen durch diese Variable zu referenzieren (*sharing*). Vor dem Kopieren dieses referenzierten Teilterms muß er erst soweit wie möglich ausgewertet werden. Die Auswertung des referenzierten Terms wird also nur dadurch angestossen, wenn der Teilterm innerhalb der Auswertung des Super-Terms benötigt wird (*call-by-need*).

2.2.2 Kontexte

Wir passen die Kontext-Definition 2.1.6 von Seite 7 an die oben beschriebene Syntax der Sprache Λ_{need} an.

Definition 2.2.2 Sei t ein beliebiger Ausdruck (oder Term) der Sprache Λ_{need}. Ein *Kontext* C ist ein Λ_{need}-Ausdruck mit einem *Loch* $[\cdot]$ und folgenden syntaktischen Bildungsgesetzen:

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

Die Symbole $C, D, \ldots$ bezeichnen Kontexte. Der Ausdruck $C[\ t\ ]$ entspricht dem Ausdruck, der nach der Ersetzung des Lochs $[\cdot]$ durch den Ausdruck t resultiert. Der Ausdruck t kann wiederum ein Kontext sein, so daß Ausdrücke wie z. B. $C[D]$ ebenfalls möglich sind.

Zum besseren Verständnis von Kontexten soll ein kleines Beispiel dienen:

Beispiel 2.2.1 Gegeben seien der Kontext $C \equiv \texttt{let}\ x\ \texttt{=}\ \lambda z.z\ \texttt{in}\ \lambda y.[\cdot]$ und der Ausdruck $t = xy$. Der Ausdruck $C[\ t\ ]$ entspricht dann dem Ausdruck $\texttt{let}\ x\ \texttt{=}\ \lambda z.z\ \texttt{in}\ \lambda y.xy$. Auch hier gilt die Eigenschaft des Kontexts, daß die in t freien Variablen x und y durch die Einsetzung in $C[\ t\ ]$ gebunden werden.

Für unsere Kalküle werden wir in den folgenden Kapiteln noch andere Kontext-Definitionen vorstellen, durch die Ausdrücke unserer Kernsprache einfacher und allgemeiner dargestellt werden können.

2.2.3 Reduktion des Kalküls λ_{need}

Nachdem wir die Sprache Λ_{need} und Kontexte vorgestellt haben, kommen wir nun zu den Reduktionsregeln, die die Auswertung des Kalküls λ_{need} beschreiben.

Definition 2.2.3 (λ_{need}) Seien C, D beliebige Kontexte. Der *Kalkül* λ_{need} umfaßt folgende Reduktionsregeln:

$$
\begin{array}{lcl}
C[\ (\lambda x\,.t)\ s\ ] & \xrightarrow{lbeta} & C[\ \texttt{let}\ x = s\ \texttt{in}\ t\ ] \\
C[\ (\texttt{let}\ x = t_x\ \texttt{in}\ s)\ t\ ] & \xrightarrow{lapp} & C[\ \texttt{let}\ x = t_x\ \texttt{in}\ (s\ t)\ ] \\
C[\ \texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ s\ ] & \xrightarrow{llet} & C[\ \texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ s)\ ] \\
C[\ \texttt{let}\ x = s\ \texttt{in}\ D[\ x\ ]\ ] & \xrightarrow{cp} & C[\ \texttt{let}\ x = s\ \texttt{in}\ D[\ s'\ ]\ ],
\end{array}
$$

falls s eine *Abstraktion* oder eine *Variable* ist, und s' eine umbenannte Kopie von s ist.

Die einzelnen Regeln sind:

(1) (*lbeta*)[8] — Die Regel korrespondiert mit der β-Kontraktion des klassischen λ-Kalküls, jedoch wird der Ausdruck s nicht sofort bei allen Vorkommen von x in t ersetzt, sondern an eine neu eingeführte Variable gebunden.

(2) (*lapp*)[9] — Durch diese Regel werden `let`-gebundene Ausdrücke aus dem Funktionsteil einer Applikation entfernt.

(3) (*llet*)[10] — Diese Regel benutzt die Assoziativ-Regel, um links-geschachtelte `let`-Ausdrücke in rechts-geschachtelte umzuwandeln.

(4) (*cp*)[11] — Diese Regel ist die wichtigste Regel des Kalküls, da sie die *call-by-need*-Eigenschaft darstellt. Ein Ausdruck, der nicht mehr weiter reduziert werden kann, wird in alle Referenzen x eingesetzt. Also auch nur, wenn der an die Variable gebundene Term wirklich benötigt wird.

2.3 Funktionale Programmiersprachen

Wie schon im vorhergehenden Kapitel erwähnt, gilt der λ-Kalkül als Vorlage für funktionale Programmiersprachen wie LISP, ML, Miranda, Scheme, Haskell und Clean, um nur einige bekannte zu nennen. Die Idee dieser Sprachen ist, wie der Name schon sagt, Programme nicht wie bei imperativen Programmiersprachen als Sequenzen von Anweisungen zu bearbeiten, sondern das Problem in kleine Teile zu zerlegen und diese durch Funktionen darzustellen. Meist reicht es in funktionalen Programmiersprachen, das Problem zu formalisieren, um das Problem zu lösen. Die Vorteile dieser Vorgehensweise sind in [Hug89] sehr gut dargestellt, eine umfassendere Einführung findet sich in [BW88].

[8] Im Original I für *introduce*, engl. für einführen.
[9] Im Original C für *commute*, engl. für vertauschen.
[10] Im Original A für *associate*.
[11] Im Original V für *value*, engl. für Wert.

Die oben genannten funktionalen Programmiersprachen unterscheiden sich dahingehend, daß sie verschiedene I/O-Konzepte und entsprechende Unterschiede in der Auswertung der Funktionen bzw. Programme besitzen. Eine kurze und informative Übersicht findet sich in [Kut00].

2.3.1 *Call-by-value*, *call-by-name* und *call-by-need*

Der Hauptunterschied funktionaler Programmiersprachen ist die Auswertung des Programms bzw. von Termen. In [MOTW95] und [MOW98] werden diese Strategien näher untersucht. Wir wollen nur kurz diese unterschiedlichen Auswertungsstrategien andeuten.

call-by-value

Bei der *call-by-value*-Auswertung werden beim Aufruf einer Funktion erst die Argumente ausgewertet und dann die Funktion selbst. Als Vorteil erscheint, daß schon einmal ausgewertete Argumente nicht nochmal betrachtet werden müssen. Der Nachteil liegt aber klar auf der Hand, wenn die Auswertung eines Arguments nicht terminiert, obwohl das Argument im Funktionskörper gar nicht benötigt wird. Die gesamte Funktion terminiert dann nicht. Programmiersprachen, die diese Art der Auswertung verwenden, nennt man auch *strikt*, da sie die Auswertung erzwingen.

call-by-name

Die *call-by-name*-Auswertung geht so vor, daß erst alle Vorkommen der Parameter innerhalb der Funktion durch die jeweiligen Argumente ersetzt (substituiert) werden, und dann die Auswertung der Funktion beginnt. Der Nachteil dieser Methode ist ebenso offensichtlich, da unausgewertete Argumente mehrfach in den Funktionskörper eingesetzt werden können und daher möglicherweise auch mehrfach ausgewertet werden müssen. Es ergibt sich dadurch sowohl ein Platz- als auch ein Zeit-Problem.

call-by-need

Die Auswertung *call-by-need* versucht nun den Nachteil von *call-by-name* bzgl. der Mehrfachauswertung zu kompensieren, indem mittels *sharing* nicht die Argumente direkt im Funktionskörper ersetzt werden, sondern nur eine Referenz darauf. Diese Referenz wird durch den `let`-Konstrukt realisiert (siehe Kapitel 2.2.1), und die Auswertung wird erst angestossen, wenn das Argument „benötigt“ wird. Funktionale Programmiersprachen, die per *call-by-need* auswerten, werden auch *nicht-strikt* oder auch *lazy* genannt.

2.3.2 Pure und impure funktionale Programmiersprachen

Pure funktionale Programmiersprachen erlauben nur Erweiterungen, die mit dem klassischen λ-Kalkül vereinbar sind. In diesem Zusammenhang hat der Begriff der *referenziellen Trans-*

parenz eine große Bedeutung. Ein Ausdruck t wird als „referenziell transparent“ bezeichnet, wenn jeder Teilterm von t mit seinem Wert (nach der Auswertung) ausgetauscht werden kann und sich dadurch der Wert (ebenfalls nach der Auswertung) von t nicht ändert. Alle Referenzen eines Terms besitzen also, unabhängig von der Auswertungsreihenfolge, immer denselben Wert. Seiteneffekte entstehen ja gerade, wenn zu keiner Zeit sichergestellt ist, daß eine Funktion bei gleichen Argumenten auch immer dasselbe Ergebnis liefert. Sprachen ohne diese Eigenschaft sind z. B. Scheme oder Standard ML, die Zuweisungen und *exceptions* erlauben.

Große Nachteile von puren funktionalen Programmiersprachen sind daher die Behandlung von I/O-Operationen und der sich daraus ergebende „unübliche“ Programmierstil. Ein wesentlicher Vorteil purer funktionaler Programmiersprachen ist jedoch die Freiheit von Seiteneffekten, die es erlaubt, diese Sprachen besser zu analysieren. Die in dieser Arbeit vorgestellten Kalküle basieren auf denen der puren funktionalen Programmiersprachen, werden jedoch durch eine nichtdeterministische Komponente erweitert, die Eingaben des Benutzers modellieren soll. Es wird also versucht, Eingabe (als Teil von I/O) auf Kalkül-Ebene zu repräsentieren.

2.3.3 Haskell: Eine nicht-strikte, pure funktionale Programmiersprache

Wir implementieren Jonah in der in [HAB+99a] definierten nicht-strikten funktionalen Programmiersprache Haskell[12]. Dieses Kapitel soll keine Einführung in die Programmierung in Haskell sein, sondern nur dazu beitragen, die in dieser Arbeit vorgestellten Funktionen verständlich zu halten. Weiterhin soll anhand der Beispiele der verwendete Programmierstil vorgestellt werden, da Haskell-Code bei komplizierten Funktionen sehr schnell unübersichtlich werden kann.

Algebraische Datentypen

Standardtypen von Haskell sind z. B. Boole'sche Werte (`Bool`), wie sie in der Standard-Prelude definiert sind. `False` und `True` sind die zum Datentyp `Bool` gehörigen Konstruktoren.

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

In Haskell werden Listen von algebraischen Datentypen durch die Konstruktoren *nil* (`[]`) und *cons* (`:`) definiert.

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

Eine Zeichenkette (*String*[13]) ist eine Liste von Zeichen (`Char`) und wird genauso definiert, nämlich als neuer Datentyp, der auf einen anderen aufbaut.

[12] Unter der Adresse `http://www.haskell.org` finden sich alle Informationen über Haskell: Compiler, Bibliotheken, Literatur uvm.

[13] Wir werden in dieser Arbeit weiterhin den englischen Ausdruck *String* benutzen.

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

Datentypen können zu Tupeln kombiniert werden, wobei die Datentypen verschieden sein können, z. B. `(Bool,Int,String)`.

In vielen Fällen kann der Aufruf einer Funktion fehlschlagen und somit kein Ergebnis liefern. Aus diesem Grund wird in dieser Arbeit oft vom Datentyp `Maybe` Gebrauch gemacht. Durch ihn kann man das Fehlschlagen einer Funktion beschreiben, nämlich durch den Konstruktor `Nothing`, der anzeigt, daß die Funktion kein Ergebnis liefern kann. Der Konstruktor `Just` enthält im erfolgreichen Fall das Ergebnis.

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

Typklassen

Möchte man bestimmte Funktionen für mehrere Datentypen bereitstellen, dann bildet man in Haskell dazu eine Klasse, die diese Funktionen (Methoden) bereitstellt. Beispielsweise stellt die Klasse `Eq` die zwei Funktionen `==` (gleich) und `/=` (ungleich) bereit, die beide zwei Argumente des Datentyps `a` erhalten und einen Wert vom Typ `Bool` zurückgeben.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y      = not (x == y)
  x == y      = not (x /= y)
```

Wenn wir jetzt einen neuen Datentyp definieren und dessen Werte vergleichen wollen, dann müssen wir eine Instanz der Klasse `Eq` für diesen neuen Datentyp definieren.

```
data NEWBOOL = TRUE
             | FALSE

instance Eq NEWBOOL where
  TRUE  == TRUE  = True
  FALSE == FALSE = True
  TRUE  == FALSE = False
  FALSE == TRUE  = False
```

In diesem Fall haben wir alle Möglichkeiten des Vergleichs auf Gleichheit definiert. Wir hätten auch die letzten beiden Zeilen durch `a == b = False` ersetzen können, um auszudrücken, daß alle übrigen Vergleiche `False` ergeben. Übrigens müssen keine Definitionen für die Funktion `/=` angegeben werden, da `/=` in der Klasse über `==` definiert ist.

Das ganze hätten wir uns allerdings sparen können, wenn wir bei der Definition von `NEWBOOL` den Compiler angewiesen hätten, automatisch eine Instanz der Klasse `Eq` zu erzeugen.

```
data NEWBOOL = TRUE
             | FALSE
             deriving (Eq)
```

In dieser Arbeit werden wir auch eine `Eq`-Instanz für Ausdrücke unserer verwendeten Kernsprache definieren, da die automatisch erzeugte `Eq`-Instanz nicht die Umbenennung von Variablen berücksichtigt (α-Konvertibilität). In `letrec`-Ausdrücken können nach unserer Definition die Bindungen ungeordnet vorkommen. Auch dieses Merkmal muß durch die spezielle `Eq`-Instanz realisiert werden.

Pattern matching

Wir haben im vorherigen Beispiel erwähnt, daß die `Eq`-Instanz auch folgendermaßen definiert werden kann:

```
instance Eq NEWBOOL where
  TRUE  == TRUE  = True
  FALSE == FALSE = True
  a     == b     = False
```

Das `a` und das `b` sind sogenannte *patterns*, also Muster für einen Konstruktor. In vielen Haskell-Ausdrücken, wie z. B. `let`- und `case`-Ausdrücken, können für einen Wert eines Datentyps diese Muster verwendet werden, die dann mit den entsprechenden Werten belegt werden und im weiteren Verlauf benutzt werden können.

```
f x = case x
        of Nothing -> False
           Just n  -> n < 10
```

Hier wird der Wert des Arguments `x` mit den Mustern `Nothing` und `Just n` verglichen. Ist der Wert von `x` z. B. `Just 3`, dann paßt das zweite Muster (*pattern matching*[14]) und der lokalen Variable `n` wird der Wert `3` zugewiesen, die dann überprüft werden kann. Haskell erlaubt das *pattern matching* auch in Funktionsdefinitionen, so daß obiges Beispiel auch so lauten kann:

```
f Nothing  = False
f (Just n) = n < 10
```

Der Konstruktor muß geklammert werden, da das Typ-System sonst die Anzahl der Argumente von `f` nicht ermitteln bzw. „matchen" kann. Die meisten Funktionen dieser Arbeit sind auf diese Weise definiert.

Ein Nachteil beim *pattern matching* ist, daß zwar allgemeine Muster mit den Argumenten verglichen werden können, wenn diese aber passen, kann man sie nicht so einfach weiterverarbeiten. Aus diesem Grund gibt es sogenannte *as-patterns*, die den passenden Ausdruck referenzieren und dessen Weiterverarbeitung ermöglichen, ohne alle Argumente einzeln zu deklarieren.

```
example = case x
            of Nothing    -> False
               a@(Just _) -> compute a
```
statt
```
                 (Just y) -> compute (Just y)
```

[14]engl.: Mustervergleich

List comprehensions

In Haskell können Listen auf vielfache Weise definiert werden. Die einfachste ist die Definition über sogenannte *list comprehensions*. Mit diesen Konstrukten können sowohl Listen erstellt, aber auch vorhandene Listen bearbeitet werden. Die Syntax dieser Konstrukte ist folgende:

```
expr -> [ x | qual1,...,qualn ]
qual -> pat <- expr   -- Generator
      | let decl      -- lokale Deklarationen
      | expr          -- Guards
```

Durch die Generatoren können z. B. einer Variablen nacheinander die Werte einer Liste zugeordnet werden. Mittels *guards* wird entschieden, ob das aktuelle (generierte) Element in der resultierenden Liste enthalten sein soll oder nicht.

Ein paar Beispiele zeigen, wie mit *list comprehensions* einfache Listen erstellt werden können.

```
[x + 1 | x <- [1..4]] = [2,3,4,5]

[x | x <- [1..10], x `mod` 2 == 0] = [2,4,6,8,10]

[(x,y) | x <- [1..2], y <- "abc"]
= [(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
```

Die *list comprehensions* in dieser Arbeit werden wir in einem bestimmten Layout definieren, das diese Konstrukte etwas übersichlicher darstellt:

```
example = [ (x,y)
          | x <- [1..10]
          , y <- [1..10]
          , x /= y
          ]
```

Auch wenn die Definition einer *list comprehension* die Reihenfolge der Listenelemente andeutet, so kann man sich nicht sicher sein, ob sie auch eingehalten wird. Das liegt an der Auswertungsreihenfolge von nicht-strikten Programmiersprachen, die nicht immer aus der Definition einer Funktion ersichtlich ist und für Haskell auch gar nicht explizit definiert ist.

Guards

In diesem Zusammenhang sollten noch die sogenannten *guards* Erwähnung finden, die sowohl in den *list comprehensions* auftreten können, als auch als Ersatz für (`if ... then ... else ...`)-Konstrukte fungieren können.

```
isGreenOrRed x
  | x == Green = "green"
  | x == Red   = "red"
  | otherwise  = "neither green nor red"
```

In diesem Beispiel wird nur die erste Funktionsdefinition benutzt, deren *guard* den Wert `True` liefert, wobei in Haskell `otherwise = True` definiert ist. Der Vorteil von *guards* ist die übersichtliche Darstellung mehrerer Vergleiche, die ansonsten durch verschachtelte (`if` … `then` … `else` …)-Konstrukte realisiert werden. Da innerhalb der *guards* auch Ausdrücke evaluiert werden können, bieten sie mehr Möglichkeiten als `case`-Ausdrücke, die nur Muster vergleichen können.

Der `do`-Konstrukt

Haskell benutzt monadisches I/O, welches in der traditionellen Form (siehe [Wad95]) nicht einfach zu lesen und verstehen ist. Daher stellt Haskell eine Notation bereit, die die monadischen Funktionen vor dem Programmierer „versteckt“. Dadurch ist eine an imperative Programmiersprachen angelehnte Programmierung möglich.

Das folgende Beispiel gibt einen String auf den Bildschirm aus, liest eine Eingabezeile von der Tastatur ein und gibt diese eingegebene Zeile wieder aus. Die Reihenfolge der Anweisungen ist dabei sofort ersichtlich.

```
main = do
         putStr "Name: "
         name <- getLine
         putStrLn ("Name = " ++ name)
```

Kapitel 3

Reduktion

Wir erweitern den in Kapitel 2.2 beschriebenen Kalkül λ_{need} durch Konstruktoren und einen `choice`-Konstrukt, der Nichtdeterminismus modelliert. Dieser Kalkül λ_{nd} soll uns als Grundlage zur Validierung von Reduktionsdiagrammen dienen. Eine rekursive Variante des Kalküls λ_{nd} wird in Form des Kalküls $\lambda_{nd,rec}$ vorgestellt. Zur Reduktion von Ausdrücken der Sprachen Λ_{nd} und $\Lambda_{nd,rec}$ in Normalordnung (NO[1]) benötigen wir noch ein Verfahren, das in einem Ausdruck den nächsten möglichen Redex markiert. Da wir in unserem System jedoch auch alle anderen möglichen Redexe reduzieren müssen, beschäftigt sich Kapitel 3.3 mit diesen Markierungsverfahren.

3.1 Die Reduktionskalküle $\lambda_{\mathbf{nd}}$ und $\lambda_{\mathbf{nd,rec}}$

In Programmiersprachen sollte es möglich sein, abstrakte Datentypen zu verwenden. Z. B. kann in Haskell mit der `data`-Anweisung ein neuer Datentyp (auch aufbauend auf andere Datentypen und sogar rekursiv) definiert werden. Weiterhin sollte man diese Datentypen während des Programmablaufs unterscheiden können. Dazu erweitern wir den Kalkül λ_{need}.

3.1.1 Die Sprachen $\Lambda_{\mathbf{nd}}$ und $\Lambda_{\mathbf{nd,rec}}$

Als zusätzliche Sprachelemente fügen wir zur Sprache Λ_{need} Konstruktoren C, einen `case`-Ausdruck sowie einen `choice`-Konstrukt, der *erratic*-Nichtdeterminismus modelliert, hinzu. Die Alternativen eines `case`-Ausdrucks bestehen aus einem *pattern*[2] und einem diesem zugewiesenem Ausdruck. Das *pattern* stellt eine pure Konstruktoranwendung dar, die als Muster für den ausgewerteten Ausdruck dient.

[1] Wir werden in dieser Arbeit die Auswertung in Normalordnung mit dem Kürzel *NO* bezeichnen. Abgeleitete Ausdrücke sind beispielsweise *NO-Redex* und *NO-Reduktion.*

[2] engl. Muster.

Definition 3.1.1 Die *Sprachen* $\Lambda_{\mathbf{nd}}$ und $\Lambda_{\mathbf{nd,rec}}$ bestehend aus Variablen, Konstruktoren, Termen und Alternativen, sind folgendermaßen strukturiert:

$$
\begin{array}{rlcl}
\text{Variablen:} & V & ::= & \{x, y, \dots\} \\
\text{Konstruktoren:} & C & ::= & \{c_A, c_B, \dots\} \\
\text{Terme:} & E & ::= & V \\
 & & | & C \\
 & & | & (\texttt{choice}\ E_1\ E_2) \\
 & & | & \lambda V.E \\
 & & | & (E_1\ E_2) \\
(\lambda_{nd}) & & | & (\texttt{let}\ V_1\texttt{=}E_1\ \texttt{in}\ E) \\
(\lambda_{nd,rec}) & & | & (\texttt{letrec}\ V_1\texttt{=}E_1\texttt{,}\dots\texttt{,}V_n\texttt{=}E_n\ \texttt{in}\ E) \\
 & & | & (\texttt{case}_A\ E\ \texttt{of}\ Alt_1\texttt{,}\dots\texttt{,}Alt_{|A|}) \\
\text{Alternativen:} & Alt & ::= & c\ V_1 \dots V_{ar(c)}\ \texttt{->}\ E
\end{array}
$$

Die Syntax beider Sprachen unterscheiden sich lediglich im `let`- bzw. `letrec`-Konstrukt. Wir benutzen für Abstraktionen ($\lambda x.s$) die Schreibweisen `\x.s` bzw. `\x->s` — in Anlehnung an Haskell.

Im Gegensatz zu [AFM+95] benutzen wir eine abgeänderte Form der in [MOW98] angegebenen Sprache, die auch Konstruktoranwendungen auf beliebige Terme erlaubt. Allerdings müssen wir dafür eine weitere Reduktionsregel einführen, die diese Terme aus der Konstruktoranwendung heraus abstrahiert (siehe Kapitel 3.1.4)[3]. Mittels `case`-Termen können wir nun abhängig von einer Konstruktoranwendung aus mehreren Alternativen eine ganz bestimmte auswählen. Dadurch sind bedingte Ausdrücke und natürlich die Identifikation der Datenstrukturen möglich.

Variablennamen

Während des Reduzierens eines Ausdrucks kann es vorkommen, daß aus einer freien Variable eine gebundene wird. Um solche unerwünschten Fälle zu umgehen, machen wir folgende Konvention:

Konvention 3.1.1 Alle durch Abstraktionen und `let(rec)`-Ausdrücke eingeführten Bindungen erfolgen durch Variablen mit eindeutigem Namen.

Beispiel 3.1.1 Der Term (`let` x `=` (`let` y `=` t_y `in` t_x) `in` y) enthält die Variable y sowohl frei als auch gebunden. Nach einer ($llet$)-Reduktion würde die Variable y nicht mehr frei sein (`let` x `=` t_x `in let` y `=` t_y `in` y).

Konstruktoren

In dieser Arbeit werden in einigen Beispielen algebraische Datentypen bzw. deren Konstruktoren verwendet, die in einer Datei definiert werden können (siehe Anhang B).

[3]Wir werden sehen, daß dadurch nur Konstruktoranwendungen kopiert werden, die nicht weiter ausgewertet werden können.

Boole'sche Werte:

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

Listen können entweder leer sein (`Nil`), oder aus einem Listenelement (`x`) und einer Restliste (`xs`) bestehen, wobei in unserer Sprache keine Typ-Überprüfung möglich ist und somit nicht die in Haskell gebräuchliche Syntax verwendet wird.

```
data List = Nil
          | Cons x xs;
```

Zahlen beschreiben wir durch Peano-Zahlen:

```
data Nat  = Zero
          | Succ x;
```

3.1.2 Reduktionskontext und Oberflächenkontext

Die im vorigen Kapitel angegebenen Definitionen für Kontexte (siehe Definition 2.2.2 und siehe Definition 2.2.2) erweitern wir um die neuen Sprachstrukturen von Λ_{nd} bzw. $\Lambda_{nd,rec}$.

Definition 3.1.2 Ein Kontext in den Sprachen Λ_{nd} bzw. $\Lambda_{nd.rec}$ ist folgendermaßen definiert:

$$
\begin{array}{lll}
C & ::= & [\,] \mid C\ E \mid E\ C \mid (\backslash x\,.C) \\
 & & \mid (\texttt{choice}\ C\ E) \mid (\texttt{choice}\ E\ C) \\
 & & \mid (\texttt{case}_A\ C\ \texttt{of}\ alts) \mid (\texttt{case}_A\ E\ \texttt{of}\ \ldots,(p\ \texttt{->}\ C),\ldots) \\
 & & \mid (\texttt{let}\ \ldots\ \texttt{in}\ C) \mid (\texttt{let}\ x\ \texttt{=}\ C\ \texttt{in}\ E) \\
 & & \text{bzw.} \\
 & & \mid (\texttt{letrec}\ \ldots\ \texttt{in}\ C) \mid (\texttt{letrec}\ \ldots,x_i\ \texttt{=}\ C,\ldots\ \texttt{in}\ E)
\end{array}
$$

Wir benutzen im Folgenden die Notation $\backslash x\,.t$ für Abstraktionen.

Definition 3.1.3 Die *Kontext-Klassen* R, R^- und L_R sind für den Kalkül λ_{nd} wie folgt definiert:

$$
\begin{array}{lll}
R^- & ::= & [\,] \mid R^-\ E \mid (\texttt{case}_A\ R^-\ \texttt{of}\ alts) \\
R & ::= & R^- \mid (\texttt{let}\ x\ \texttt{=}\ E\ \texttt{in}\ R) \mid (\texttt{let}\ x\ \texttt{=}\ R^-\ \texttt{in}\ R[x]) \\
L_R & ::= & [\,] \mid (\texttt{let}\ x\ \texttt{=}\ E\ \texttt{in}\ L_R)
\end{array}
$$

Ausdrücke der Kontext-Klasse R sind *Reduktionskontexte*.

Definition 3.1.4 Die *Kontext-Klassen* R, R^- und L_R sind für den Kalkül $\lambda_{nd,rec}$ wie folgt definiert:

$$
\begin{array}{lll}
R^- & ::= & [\,] \mid R^-\ E \mid (\texttt{case}_A\ R^-\ \texttt{of}\ alts) \\
R & ::= & R^- \mid (\texttt{letrec}\ x_1\ \texttt{=}\ E_1,\ldots,x_n\ \texttt{=}\ E_n\ \texttt{in}\ R^-) \\
 & & \mid (\texttt{letrec}\ x_1\ \texttt{=}\ R^-[\,],\ldots,x_j\ \texttt{=}\ R^-[x_{j-1}],\ldots\ \texttt{in}\ R^-[x_j]) \\
L_R^0 & ::= & [\,] \\
L_R^n & ::= & (\texttt{letrec}\ x_1\ \texttt{=}\ E_1,\ldots,x_n\ \texttt{=}\ E_n\ \texttt{in}\ [\,])
\end{array}
$$

Ausdrücke der Kontext-Klasse R sind *Reduktionskontexte*.

Um die kontextuelle Gleichheit für einige Reduktionen zu zeigen, benötigen wir eine Einschränkung der Definition 3.1.2, so daß der Kontext innerhalb einer Abstraktion kein Loch enthalten darf.

Definition 3.1.5 Ein *Oberflächenkontext* definiert einen Ausdruck, der kein Loch innerhalb einer Abstraktion enthält:

S ::= [] | S E | E S
| (`choice` S E) | (`choice` E S)
| (`case`$_A$ S `of` $alts$) | (`case`$_A$ E `of` ...,(p `->` S),...)
| (`let` ... `in` S) | (`let` x `=` S `in` E)
bzw.
| (`letrec` ... `in` S) | (`letrec` ...,x_i `=` S,... `in` E)

3.1.3 Schwache Kopfnormalform

Definition 3.1.6 (WHNF) Eine *Schwache Kopfnormalform (oder WHNF[4])* ist ein Term der Form $t \equiv L_R[t']$, wobei t' eine Abstraktion ist. Wenn ein Term eine WHNF ist, dann sagen wir auch, daß er „in“ WHNF ist.

Lemma 3.1.1 Für jeden Term t gilt:

- Wenn t einen NO-Redex enthält, dann ist dieser NO-Redex eindeutig.
- Eine NO-Reduktion ohne (nd)-Redexe ist eindeutig.

Lemma 3.1.2 Ein Term in WHNF hat keine NO-Reduktion, d. h. er kann nicht weiter reduziert werden.

Definition 3.1.7 (CWHNF) Eine *Schwache Konstruktor-Kopfnormalform (CWHNF[5])* ist ein Term der Form $t \equiv L_R[t']$, wobei t' eine pure Konstruktoranwendung ist. Wenn ein Term eine CWHNF ist, dann sagen wir auch, daß er „in“ CWHNF ist.

Definition 3.1.8 (VWHNF) Eine *Schwache Variablen-Kopfnormalform (VWHNF[6])* ist ein Term der Form $t \equiv L_R[t']$, wobei t' eine Variable ist. Wenn ein Term eine VWHNF ist, dann sagen wir auch, daß er „in“ VWHNF ist.

Einige Terme unserer Sprache besitzen keinen Redex bzw. keine Reduktion zur WHNF. Wir können sie in folgende Kategorien unterteilen:

1. Terme mit Typ-Fehler, z. B. (`case` $\lambda x . x$ `of` ...).
2. Terme, die sich schon in einer WHNF (CWHNF oder VWHNF) befinden wie $\lambda x . x$ oder `Cons True Nil`.

[4] engl. weak head normal form
[5] engl. constructor weak head normal form
[6] engl. variable weak head normal form

3. Im rekursiven Kalkül kann ein Term nicht reduziert werden, wenn die Reduktion niemals enden kann, wie in dem Term $\texttt{letrec}x\ \texttt{=}\ x\ \texttt{in}\ x$.

Wenn wir einen Redex markieren wollen, dann müssen wir diese Fälle erkennen und ausfiltern. Der in Kapitel 5 vorgestellte Termgenerator versucht solche Terme erst gar nicht zu generieren, was bis zu einem gewissen Grad — bzgl. der Termstruktur — auch funktioniert.

3.1.4 Der Kalkül λ_{nd}

Die Erweiterung des in Kapitel 2.2 beschriebenen Kalküls λ_{need} besteht aus fünf neuen Reduktionsregeln. Zusätzlich muß die (cp)-Regel für das Kopieren einer Konstruktoranwendung erweitert werden.

Definition 3.1.9 (λ_{nd}) Der *nichtdeterministische Kalkül* λ_{nd} umfaßt folgende Reduktionsregeln:

$C[\ (\lambda x\,.t)\ s\ ]$	$\xrightarrow{lbeta}$	$C[\ \texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ t\ ]$
$C[\ c\ s_1 ... s_i ... s_n\ ]$	$\xrightarrow{abs}$	$C[\ \texttt{let}\ y\ \texttt{=}\ s_i\ \texttt{in}\ (c\ s_1 ... y ... s_n)\ ]$
falls $s_1 ... s_{i-1}$ Variablen und s_i **keine** Variable.		
$C[\ (\texttt{let}\ x\ \texttt{=}\ t_x\ \texttt{in}\ s)\ t\ ]$	$\xrightarrow{lapp}$	$C[\ \texttt{let}\ x\ \texttt{=}\ t_x\ \texttt{in}\ (s\ t)\ ]$
$C[\ \texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ D[\ x\ ]\ ]$	$\xrightarrow{cp}$	$C[\ \texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ D[\ s\ ]\ ]$
falls s Abstraktion oder pure Konstruktoranwendung.		
$C[\ \texttt{let}\ x\ \texttt{=}\ (\texttt{let}\ y\ \texttt{=}\ t_y\ \texttt{in}\ t_x)\ \texttt{in}\ s\ ]$	$\xrightarrow{llet}$	$C[\ \texttt{let}\ y\ \texttt{=}\ t_y\ \texttt{in}\ (\texttt{let}\ x\ \texttt{=}\ t_x\ \texttt{in}\ s)\ ]$
$C[\ \texttt{case}_A\ (\texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ t)\ \texttt{of}\ Alts\ ]$	$\xrightarrow{lcase}$	$C[\ \texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ (\texttt{case}_A\ t\ \texttt{of}\ Alts)\ ]$
$C[\ \texttt{case}_A\ (c_{A,i}\ \vec{y})\ \texttt{of}\ \ldots, c_{A,i}\ \vec{x_i}\,.t_i, \ldots\ ]$	$\xrightarrow{case}$	$C[\ \texttt{let}\ x_{i,1}\ \texttt{=}\ y_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_{i,j}\ \texttt{=}\ y_j\ \texttt{in}\ t_i\ ]$
mit $ar(c_{A,i}) = n$.		
$C[\ \texttt{choice}\ s\ t\ ]$	$\xrightarrow{ndl}$	$C[\ s\ ]$
	$\xrightarrow{ndr}$	$C[\ t\ ]$

Die einzelnen Regeln sind:

(1) (*lbeta*) — Die Regel korrespondiert mit der β-Kontraktion des klassischen λ-Kalküls, jedoch wird der Ausdruck t nicht sofort bei allen Vorkommen von x in s ersetzt, sondern an die Variable x gebunden. Erst in weiteren Auswertungsschritten zeigt sich, ob t überhaupt benötigt wird und ausgewertet werden muß. Durch dieses *sharing* genannte Prinzip können viele unnötige Rechenschritte gespart werden.

(2) (*lapp*) — Diese Regel entfernt `let`-Ausdrücke aus dem Funktionsteil einer Applikation.

(3) (*llet*) — Ein `let`-Ausdruck innerhalb einer `let`-Bindung wird aus dem Bindungsteil *heraus geschoben*. Wegen unserer Konvention, Variablen nur mit eindeutigen Namen zu versehen, ergeben sich hierbei keine Konflikte bzgl. des Variablen-Skopus.

(4) (*abs*) — Beim Kopieren von Konstruktoranwendungen kann es passieren, daß unausgewertete Argumente später, wenn sie benötigt werden, mehrfach ausgewertet werden.

Um dies zu vermeiden, abstrahieren wir die Argumente, so daß die Argumente eines Konstruktors beim Kopieren nur noch Variablen sind.

(5) (*cp*) — Wegen der *call-by-need*-Auswertung können nur Abstraktionen und pure Konstruktoranwendungen (siehe (*abs*)) kopiert werden.

(6) (*lcase*) — Wie bei Regel (*llet*) wird hier ein `let`-Term nach *außen* geschoben. Namenskonflikte können auch hier nicht auftreten.

(7) (*case*) — Bei dieser Regel wird die (pure) Konstruktoranwendung mit allen Termen in einer Menge von Alternativen mittels Mustervergleich (= *pattern matching*) verglichen. Der `case`-Ausdruck wird durch die passende Alternative ersetzt, wobei die Argumente der puren Konstruktoranwendung, die jetzt nur noch Variablen sind, an die Variablen der Alternative gebunden werden.

(8) (*ndl*) und (*ndr*) — der `choice`-Konstrukt modelliert eine Benutzereingabe (*erratic choice*). Einer von beiden Teiltermen muß ausgewählt werden. Die Vereinigung beider Reduktionsregeln nennen wir (*nd*).

Da dieser vorgestellte Kalkül nicht rekursiv ist, d. h. Rekursion nur über Fixpunkt-Kombinatoren realisiert werden kann, werden wir nun einen neuen Kalkül vorstellen, der durch seine syntaktische Struktur Rekursion gewährleistet.

3.1.5 Der Kalkül $\lambda_{nd,rec}$

Durch die Einführung eines rekursiven `let`-Ausdrucks (`letrec`) müssen die Reduktionsregeln (*cp*), (*llet*) und (*case*) angepaßt werden. Die übrigen Regeln bleiben bis auf die Umbenennung des `let` in `letrec` erhalten.

Definition 3.1.10 Der *nichtdeterministische Kalkül* $\lambda_{nd,rec}$ umfaßt folgende Reduktionsregeln:

$C[\ (\lambda x\,.t)\ s\ ]$	$\xrightarrow{lbeta}$	$C[\ \mathtt{letrec}\ x\ \mathtt{=}\ s\ \mathtt{in}\ t\ ]$
$C[\ c\ s_1 ... s_i ... s_n\ ]$ falls $s_1 ... s_{i-1}$ Variablen und s_i **keine** Variable.	$\xrightarrow{abs}$	$C[\ \mathtt{letrec}\ y\ \mathtt{=}\ s_i\ \mathtt{in}\ (c\ s_1 ... y ... s_n)\ ]$
$C[\ (\mathtt{letrec}\ X\ \mathtt{in}\ s)\ t\ ]$	$\xrightarrow{lapp}$	$C[\ \mathtt{letrec}\ X\ \mathtt{in}\ (s\ t)\ ]$
$C[\ \mathtt{letrec}\ ...\mathtt{,}x_i\ \mathtt{=}\ s_i\mathtt{,}...\ \mathtt{in}\ D[\ x_i\ ]\ ]$ falls s_i Abstraktion oder pure Konstruktoranwendung.	$\xrightarrow{cp}$	$C[\ \mathtt{letrec}\ ...\mathtt{,}x_i\ \mathtt{=}\ s_i\mathtt{,}...\ \mathtt{in}\ D[\ s_i'\ ]\ ]$
$C[\ \mathtt{letrec}\ ...\mathtt{,}x_i\ \mathtt{=}\ s_i\mathtt{,}...\mathtt{,}x_j\ \mathtt{=}\ D[x_i]\mathtt{,}...\ \mathtt{in}\ s\ ]$ falls s_i Abstraktion oder pure Konstruktoranwendung.	$\xrightarrow{cp}$	$C[\ \mathtt{letrec}\ ...\mathtt{,}x_j\ \mathtt{=}\ D[s_i']\mathtt{,}...\ \mathtt{in}\ s\ ]$
$C[\ \mathtt{letrec}\ X\ \mathtt{in}\ (\mathtt{letrec}\ Y\ \mathtt{in}\ s)\ ]$ $X = \{x_1\ \mathtt{=}\ s_1\mathtt{,}...\mathtt{,}x_n\ \mathtt{=}\ s_n\}$ und $Y = \{y_1\ \mathtt{=}\ t_1\mathtt{,}...\mathtt{,}y_m\ \mathtt{=}\ t_m\}$.	$\xrightarrow{lletr1}$	$C[\ \mathtt{letrec}\ (X \cup Y)\ \mathtt{in}\ s\ ]$
$C[\ \mathtt{letrec}\ ...\mathtt{,}x_k\ \mathtt{=}\ (\mathtt{letrec}\ Y\ \mathtt{in}\ s_k)\mathtt{,}...\ \mathtt{in}\ s\ ]$ $X = \{x_1\ \mathtt{=}\ s_1\mathtt{,}...\mathtt{,}x_n\ \mathtt{=}\ s_n\}$ und $Y = \{y_1\ \mathtt{=}\ t_1\mathtt{,}...\mathtt{,}y_m\ \mathtt{=}\ t_m\}$.	$\xrightarrow{lletr2}$	$C[\ \mathtt{letrec}\ (X \cup Y)\ \mathtt{in}\ s\ ]$
$C[\ \mathtt{case}_A\ (\mathtt{letrec}\ X\ \mathtt{in}\ s)\ \mathtt{of}\ Alts\ ]$	$\xrightarrow{lcase}$	$C[\ \mathtt{letrec}\ X\ \mathtt{in}\ (\mathtt{case}_A\ s\ \mathtt{of}\ Alts)\ ]$
$C[\ \mathtt{case}_A\ (c_{A,i}\ y_1 ... y_n)\ \mathtt{of}\ ...\mathtt{,}c_{A,i}\ \vec{x_i}\,\mathtt{.}t_i\mathtt{,}...\ ]$ mit $ar(c_{A,i}) = n$.	$\xrightarrow{case}$	$C[\ \mathtt{letrec}\ x_{i,1}\ \mathtt{=}\ y_1\mathtt{,}...\mathtt{,}x_{i,n}\ \mathtt{=}\ y_n\ \mathtt{in}\ t_i\ ]$
$C[\ \mathtt{choice}\ s\ t\ ]$	$\xrightarrow{ndl}$	$C[\ s\ ]$
	$\xrightarrow{ndr}$	$C[\ t\ ]$

Die einzelnen Regeln sind:

(1) (*lbeta*) — wie bei λ_{nd}

(2) (*abs*) — wie bei λ_{nd}

(3) (*lapp*) — wie bei λ_{nd}

(4) (*cp*) — die `letrec`-gebundenen Variablen können beim rekursiven Kalkül auch innerhalb der `letrec`-Bindungen kopiert werden.

(5) (*lletr1*) — weil der `letrec`-Konstrukt mehrere Bindungen enthalten kann, können alle Bindungen in ein einziges `letrec` *verschmolzen* werden[7].

(6) (*lletr2*) — wie bei *(lletr2)* werden alle `letrec`-Bindungen zusammengezogen.

(7) (*lcase*) — wie bei λ_{nd}

(8) (*case*) — im Unterschied zur (*case*)-Regel des nicht-rekursiven Kalküls können die Bindungen der *pattern*-Variablen direkt in einen `letrec`-Ausdruck „gepackt“ werden. Eigentlich ist die (*case*)-Regel durch *versteckte* (*lletr1*)-Regeln erweitert worden.

(9) (*ndl*) und (*ndr*) — wie bei λ_{nd}

Bevor wir die Implementierung der beiden Kalküle bzw. deren Reduktionsmechanismus vorstellen, befassen wir uns mit der Darstellung der Kernsprache in Haskell-Code und beschreiben einen Parser für diese Sprache.

[7] Im nicht-rekursiven Kalkül ist dies nicht möglich, da sonst rekursive Abhängigkeiten entstehen würden.

3.2 Die Kernsprache

Wir werden nun eine Kernsprache vorstellen, die den oben vorgestellten Kalkülen λ_{nd} und $\lambda_{nd,rec}$ zugrunde liegt.

3.2.1 Die Syntax der Kernsprache

Die verwendete Kernsprache (siehe Abbildung 3.1, Seite 28) besteht aus Variablen, Konstruktoren, Abstraktionen, Applikationen, `let(rec)`-, `case`- und `choice`-Ausdrücken. Die algebraischen Datentypen sind nicht getypt, d. h. es wird in dieser Arbeit keine Typüberprüfung der erzeugten bzw. eingegebenen Ausdrücke verwendet (siehe Kapitel 3.1.1, Seite 20).

Darstellung der Kernsprache in Haskell

Zur Darstellung der Kernsprache in Haskell verwenden wir den abstrakten Datentyp *CoreExpr*, der auch in anderen Realisierungen so ähnlich implementiert ist ([PJ87],[PJL91]).

```
data CoreExpr
     = EVar    Mark VName                         -- Variable
     | EChoice Mark          CoreExpr  CoreExpr   -- choice-Ausdruck
     | ECons   Mark CName    CArity               -- Konstruktoranwendung
     | EComb   Mark VName    [VName]   CoreExpr   -- Kombinator
     | ELam    Mark VName    CoreExpr             -- λ-Ausdruck
     | EApp    Mark CoreExpr CoreExpr             -- Anwendung
     | ELet    Mark LRec     [Binding] CoreExpr   -- let(rec)-Ausdruck
     | ECase   Mark CType    CoreExpr  [CaseAlt]  -- case-Ausdruck
     deriving (Ord)
```

Die Haskell-Typen für Variablen-, Konstruktor- und Typennamen sind folgendermaßen definiert:

```
type VName    = String                         -- Variablenname

type CType    = String                         -- Datentypname

type CName    = String                         -- Konstruktorname

type CArity   = Int                            -- Stelligkeit eines Konstruktors

type LRec     = Bool                           -- Rekursives Let

type Binding  = (VName, CoreExpr)              -- let(rec)-Bindung

type CaseAlt  = (CName, [VName], CoreExpr) -- Case-Alternative

type CoreData = (CType, [(CName, CArity)]) -- Abstrakter Datentyp
```

Während des Parsens, des Markierens der Normalordnung und der Reduktion müssen Teilausdrücke eines Ausdrucks immer wieder mit speziellen Markierungen versehen werden. Diese werden durch den leicht zu erweiternden Datentyp `Mark` realisiert.

```
data Mark
     = Unmarked                  -- ohne Markierung
     | Marked   String           -- Markierung
     | Redex    ReductionType -- Redex-Markierung
     | WHNF                      -- Weak Head Normal Form
     | VWHNF                     -- Variable Weak Head Normal Form
     | CWHNF                     -- Constructor Weak Head Normal Form
     deriving (Ord, Eq, Show)
```

3.2.2 Der Parser für die Kernsprache

Zum Erzeugen des Parsers für unsere Kernsprache benutzen wir den Parser-Generator *lucky*[8] von Norbert Klose. Die in [Klo97] beschriebene Grammatikdatei ist für unsere Kernsprache wie folgt aufgebaut:

Der Code in geschweiften Klammern wird direkt in die erzeugte Haskell-Datei übernommen. Hier werden die benötigten Module geladen und die verwendeten Datentypen definiert.

```
{

module ExpressionParser (parseCoreExpr, parseCoreProgram) where

import Char
import List
import Utils
import Expression
```

Die Schlüsselwörter der Sprache $\Lambda_{nd(,rec)}$ werden in der Funktion `lexer` identifiziert und in den Datentyp Lexem gewandelt. Für jedes Schlüsselwort wird ein Konstruktor definiert.

```
data Lexem = LexemVar VName
           | LexemCons CName
           | LexemLBracket
           | LexemRBracket
           | LexemLBracket1
           | LexemRBracket1
           | LexemLBracket2
           | LexemRBracket2
           | LexemLT
           | LexemGT
           | LexemBar
           | LexemDash
           | LexemBackslash
           | LexemDot
           | LexemComma
           | LexemSemicolon
```

[8] Da das gesamte Programm in Haskell 98 implementiert wurde, benutzen wir die von Marko Schütz angepaßte Version des *lucky*, die die Änderungen bzgl. der Monaden berücksichtigt.

Programm	$prog$	$\rightarrow$	$progitem_1$; … ; $progitem_n$	$n \geq 1$
	$progitem$	$\rightarrow$	$data$	
		$\mid$	sk	
		$\mid$	$expr$	
Datentyp	$data$	$\rightarrow$	`data` con = $constr_1$ $\mid$ … $\mid$ $constr_n$	$n \geq 1$
Superkombinator	sk	$\rightarrow$	$var\ var_1 \ldots var_n$ = $expr$	$n \geq 0$
Ausdruck	$expr$	$\rightarrow$	var	Variable
		$\mid$	$constr$	Konstruktor
		$\mid$	`choice` $expr_1\ expr_2$	`choice`-Konstrukt
		$\mid$	$expr_1\ expr_2$	Anwendung
		$\mid$	`\` var `.` $expr$	Abstraktion
		$\mid$	`\` var `->` $expr$	
		$\mid$	`let` var = $expr_1$ `in` $expr_2$	`let`(`rec`)-Ausdruck
		$\mid$	`case` $expr$ `of` $alts$	`case`-Ausdruck
		$\mid$	`[]`	Loch
		$\mid$	`(` $expr$ `)`	Geklammerter Ausdruck
Alternative	$alts$	$\rightarrow$	alt_1 , … , alt_n	$n \geq 1$
	alt	$\rightarrow$	pat `->` $expr$	
	pat	$\rightarrow$	$constr\ var_1 \ldots var_n$	$n \geq 0$
Variable	var	$\rightarrow$	$alpha\ char_1 \ldots char_n$	$n \geq 0$
Konstruktor	$constr$	$\rightarrow$	$capital\ char_1 \ldots char_n$	$n \geq 0$
Zeichen	$char$	$\rightarrow$	$alpha \mid capital \mid digit \mid special$	
	$alpha$	$\rightarrow$	`a` $\mid \ldots \mid$ `z`	
	$capital$	$\rightarrow$	`A` $\mid \ldots \mid$ `Z`	
	$digit$	$\rightarrow$	`0` $\mid \ldots \mid$ `9`	
	$special$	$\rightarrow$	`_` $\mid$ `'`	

Abbildung 3.1: Die Syntax der Kernsprache

```
           | LexemEqual
           | LexemData
           | LexemLet
           | LexemLetrec
           | LexemIn
           | LexemCase
           | LexemOf
           | LexemChoice
```

Ein Token besteht aus dem Tripel *(Lexem, Zeile, Spalte)*, so daß beim Auftreten eines Fehlers die Zeilen- und Spaltennummer des Eingabe-Textes ausgegeben werden kann.

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

}
```

Nun muß noch für jedes *lucky*-Token der entsprechende Haskell-Datentyp definiert werden.

```
%name parser
%tokentype { Token }
%token VarId       { (LexemVar   $$ , _, _) }
       ConsName    { (LexemCons  $$ , _, _) }
       '('         { (LexemLBracket , _, _) }
       ')'         { (LexemRBracket , _, _) }
       '['         { (LexemLBracket1, _, _) }
       ']'         { (LexemRBracket1, _, _) }
       '{'         { (LexemLBracket2, _, _) }
       '}'         { (LexemRBracket2, _, _) }
       '<'         { (LexemLT       , _, _) }
       '>'         { (LexemGT       , _, _) }
       '|'         { (LexemBar      , _, _) }
       '-'         { (LexemDash     , _, _) }
       '\'         { (LexemBackslash, _, _) }
       '.'         { (LexemDot      , _, _) }
       ','         { (LexemComma    , _, _) }
       ';'         { (LexemSemicolon, _, _) }
       '='         { (LexemEqual    , _, _) }
       data        { (LexemData     , _, _) }
       let         { (LexemLet      , _, _) }
       letrec      { (LexemLetrec   , _, _) }
       in          { (LexemIn       , _, _) }
       case        { (LexemCase     , _, _) }
       of          { (LexemOf       , _, _) }
       choice      { (LexemChoice   , _, _) }
```

Mit zwei aufeinander folgenden Prozentzeichen beginnt der Teil der Grammatikdatei, der die Kernsprache definiert. Ein Kernsprachenprogramm besteht aus zwei Teilen: den Definitionen der algebraischen Datentypen und den Superkombinator-Definitionen. Sowohl die Datentyp- als auch die Superkombinator-Definitionen werden mit einem Semikolon getrennt.

```
%%

Program  :: { ([CoreData], [CoreExpr]) }
          : Datatype                       { ([$1],[]  )               }
          | Datatype ';'                   { ([$1],[]  )               }
          | Datatype ';' Program           { let (d,e)=$3 in ($1:d,e)  }
          | Expr                           { ([]  ,[$1])               }
          | Expr     ';'                   { ([]  ,[$1])               }
          | Expr     ';' Program           { let (d,e)=$3 in (d,$1:e)  }
```

Die Syntax der Datentypen sieht folgendermaßen aus:

```
Datatype :: { CoreData }
          : data ConsName '=' Datas        { ($2, $4)                  }

Datas    :: { [(CName, CArity)] }
          : Data                           { [$1]                      }
          | Data '|' Datas                 { $1 : $3                   }

Data     :: { (CName, CArity) }
          : ConsName                       { ($1, 0)                   }
          | ConsName Vars                  { ($1, length $2)           }
```

Um beim Parsen der Ausdrücke schon *currying* zu erreichen, muß die Funktion `makeApp` zwei aufeinander angewandte Ausdrücke von links klammern.

```
Expr     :: { CoreExpr }
          : Comb                           { $1                        }
          | Exp                            { $1                        }

Exp      :: { CoreExpr }
          : ExpL Exp                       { makeApp $1 $2             }
          | ExpL                           { $1                        }
```

Ein Ausdruck besteht aus den oben genannten Sprach-Elementen. Zusätzlich erlauben wir für Abstraktionen die Schreibweise (`\x -> ...`). Innerhalb des Ausdrucks kann sich auch ein Loch (`[]`) befinden, das wir intern als Variable mit leerem Namen definieren.

```
ExpL     :: { CoreExpr }
          : VarId                          { checkVariable    $1       }
          | choice ExpL ExpL               { EChoice Unmarked $2 $3    }
          | ConsName                       { ECons   Unmarked $1 0     }
          | '\' VarId '.' Exp              { ELam    Unmarked $2 $4    }
          | '\' VarId '-' '>' Exp          { ELam    Unmarked $2 $5    }
          | let       Bindings     in Exp  { makeLet    $2 $4          }
          | letrec '{'          '}' in Exp { makeLetrec [] $5          }
          | letrec '{' Bindings '}' in Exp { makeLetrec $3 $6          }
          | letrec    Bindings     in Exp  { makeLetrec $2 $4          }
          | case Exp of CaseAlts           { ECase   Unmarked "" $2 $4 }
          | '[' ']'                        { EVar    VWHNF    ""       }
          | '(' Exp ')'                    { markApp          $2       }
```

Um neue Funktionen in der *Prelude*-Datei zu definieren, ist es einfacher sie als Kombinator zu beschreiben. Die Kombinatoren werden in einem eigenen Datentyp definiert und später bei der Reduktion von Ausdrücken als Funktionen hinzugefügt.

```
Comb     :: { CoreExpr }
          : VarId      '=' Exp              { EComb   Unmarked $1 [] $3 }
          | VarId Vars '=' Exp              { EComb   Unmarked $1 $2 $4 }
```

Für die Argumente von Kombinatoren und `case`-Alternativen benötigen wir eine Liste von Variablen:

```
Vars     :: { [VName] }
          : VarId                           { [$1]                      }
          | VarId Vars                      { $1 : $2                   }
```

Die Bindungen von `let(rec)`-Ausdrücken können im nicht-rekursiven Fall einfach, im rekursiven Kalkül aber auch mehrfach auftreten, so daß wir eine Liste von Bindungen erhalten:

```
Bindings :: { [Binding] }
          : VarId '=' Exp                   { [($1,$3)]                 }
          | VarId '=' Exp ',' Bindings      { ($1,$3) : $5              }
```

Die Alternativen eines `case`-Ausdrucks werden ebenfalls in einer Liste gesammelt:

```
CaseAlts :: { [CaseAlt] }
          : CaseAlt                         { [$1]                      }
          | CaseAlt ',' CaseAlts            { $1 : $3                   }


CaseAlt  :: { CaseAlt }
          : ConsName      '-' '>' Exp       { ($1, [], $4)              }
          | ConsName Vars '-' '>' Exp       { ($1, $2, $5)              }
```

Es folgt wieder Haskell-Code, der mit geschweiften Klammern eingefaßt werden muß.

```
{
```

Damit Ausdrücke unserer Variablennamenkonvention 3.1.1 genügen, müssen wir die Variablen eingegebener Ausdrücke dahingehend überprüfen, daß sie nicht mit dem Präfix `new_` beginnen[9] oder als Suffix einen Unterstrich mit folgender Zahl besitzen[10].

[9] Neu eingeführte Variablen sollen mit `new_` beginnen und können durchnumeriert werden, oder enthalten einen Namensteil, der die Herkunft der neuen Variable anzeigt, z. B. `new_case_a_5`.

[10] Wenn bei kopierenden Reduktionen Teilterme kopiert werden, dann fügen wir an das Ende des Variablennamen, getrennt durch einen Unterstrich `_`, den aktuellen Zähler der Reduktion an.

```
checkVariable v
    | newVarPrefix `isPrefixOf` v
      = error $ "ExpressionParser.checkVariable: "
               ++ "variable begins with `"
               ++ newVarPrefix ++ "`."
    | checkSuffix
      = error $ "ExpressionParser.checkVariable: "
               ++ "variable ends with `_number`."
    | otherwise
      = EVar Unmarked v
    where
      (a,uscore)  = break (== '_') (reverse v)
      checkSuffix = (a /= "") && (all isDigit a) && ("_" `isPrefixOf` uscore)
```

Wie oben angedeutet muß beim Erzeugen von Kernsprachen-Ausdrücken darauf geachtet werden, daß ungeklammerte Ausdrücke von links geklammert zu interpretieren sind. Diese Aufgabe übernimmt die Funktion `makeApp`.

```
makeApp e1 (EApp Unmarked   e1' e2') = EApp Unmarked (makeApp e1 e1') e2'
makeApp e1 (EApp (Marked _) e1' e2') = EApp Unmarked e1 (EApp Unmarked e1' e2')
makeApp e1 e2                        = EApp Unmarked e1 e2
```

Umgekehrt müssen explizit geklammerte Ausdrücke markiert werden, damit ihre Klammerung nicht von `makeApp` zerstört wird. Die Markierung wird später entfernt (siehe unten).

```
markApp (EApp m e1 e2) = EApp (Marked "") e1 e2
markApp e              = e
```

Durch die Sprachdefinition an sich können wir nicht überprüfen, ob ein `let`-Ausdruck eine rekursive Bindung enthält. Aus diesem Grund müssen `let`- und `letrec`-Ausdrücke getrennt voneinander auf Wohlgeformtheit geprüft werden. Im `let`-Fall darf nur genau eine Bindung definiert werden, die nicht rekursiv ist, d. h. die gebundene Variable nicht frei im gebundenen Term vorkommt. Das `False` im `ELet`-Ausdruck markiert den `let`-Ausdruck als nicht-rekursives `let`.

```
makeLet bs e
    = let expr = ELet Unmarked False bs e
       in case bs
            of []      -> error $ "ExpressionParser.makeLet: "
                                 ++ "exactly one let binding allowed in `"
                                 ++ show expr ++ "`"
               [(v,x)] -> if v `elem` (getFreeVars x)
                             then error $ "ExpressionParser.makeLet: "
                                         ++ "recursive let binding in `"
                                         ++ show expr ++ "`"
                             else expr
               _       -> error $ "ExpressionParser.makeLet: "
                                 ++ "too many let bindings in `"
                                 ++ show expr ++ "`"
```

Im `letrec`-Fall muß lediglich geprüft werden, ob eine Variablenbindung doppelt vorkommt. Das `True` markiert den `let`-Ausdruck als rekursives `letrec`.

```
makeLetrec bs e
    = let expr = ELet Unmarked True bs e
       in if length bs == length (nub [v | (v,_) <- bs])
             then expr
             else error $ "ExpressionParser.makeLetrec: "
                          ++ "double letrec binding in `"
                          ++ show expr ++ "`"
```

Das Kernsprachenprogramm wird von der Funktion `parseCoreProgram` geparst. Dabei werden zuerst die algebraischen Datentypen und dann die Superkombinatoren erzeugt. Die Superkombinatoren werden schließlich mit der Funktion `updateADTs` (siehe Anhang A.3) bearbeitet. Die Funktion `updateADTs` entfernt alle Markierungen, die während des Parsens gesetzt wurden, ermittelt die Stelligkeit der Konstruktoren und den Typ von `case`-Ausdrücken. Dadurch werden schon beim Parsen grobe Typ-Fehler ausgeschlossen.

Im interaktiven Teil des Systems ist es möglich einzelne Ausdrücke einzugeben. Diese können mit der Funktion `parseCoreExpr` geparst werden, wobei jedoch die algebraischen Datentypen angegeben werden müssen, sofern sie nicht schon beim Start des Systems eingelesen wurden.

Bemerkung 3.2.1 Bei der Eingabe eines Terms sowohl in der *Prelude* als auch im interaktiven Modus wird die Eindeutigkeit gebundener Variablennamen (siehe Konvention 3.1.1, Seite 20) überprüft. Daher muß bei der Verwendung der Terme, die der Parser liefert, darauf geachtet werden, die Terme mit der Funktion `correctCoreExpr` zu bearbeiten.

```
parseCoreProgram adts s
    = (adts'', expressions)
        where
          adts''          = adts ++ adts'
          expressions     = [updateADTs adts'' e | e <- exprs]
          (adts', exprs) = (parser "" (lexer s))

parseCoreExpr ds = head . snd . (parseCoreProgram ds)
```

Die Eingabe muß vor dem Parsen noch in eine Liste von Tokens umgewandelt werden. Diese Aufgabe übernimmt der sogenannte *Lexer*, der hier aus Platzgründen nicht angegeben wird. Er analysiert eine Zeichenkette mittels Mustervergleich und erzeugt aus den Schlüsselwörtern der Sprache *Lexeme*, die samt Zeilen- und Spaltennummer in einem *Token* zusammengefaßt werden.

Zum Schluß noch die Funktion `luckyError`, die eine genaue Beschreibung eines aufgetretenen Fehlers ausgibt.

```
luckyError :: String -> [Token] -> a
luckyError filename []
    = case filename
        of [] -> error ("parseCoreExpr: Unexpected end of input")
           f  -> error ("parseCoreExpr: Unexpected end of file '" ++ f ++ "'")
luckyError filename ((lexem,l,c):_)
    = error (concat ["parseCoreExpr: Syntax error in "
                   , what, "(",show l,",",show c,"): '",show lexem,"'"])
        where
          what = case filename
                   of [] -> "input "
                      f  -> "file '" ++ filename ++ "'"
```

Der Programmteil der Grammatikdatei endet mit einer abschließenden Klammer.

```
}
```

3.3 Markierungsalgorithmen

Zur Auswertung bzw. Reduktion der Kernsprachenausdrücke müssen wir einen Teilausdruck identifizieren, der als nächstes reduziert werden kann. Dieser reduzierbare Ausdruck (*Redex*) wird mittels eines Markierungsalgorithmus gesucht. Wir stellen einen Algorithmus zur Markierung eines NO-Redex für beide Kalküle vor. Zur Identifikation interner Redexe definieren wir einen Algorithmus, der Terme gezielt, abhängig vom Reduktionstyp und dem Kontext, markiert.

3.3.1 Markierung eines NO-Redex

Die Ausdrücke sollen in Normalordnung ausgewertet werden. Dazu muß vor jedem Reduktionsschritt ein Redex markiert werden. Zur Markierung eines solchen NO-Redex verwenden wir folgenden Markierungsalgorithmus.

Definition 3.3.1 (Algorithmus *MarkNO*) Der *NO-Redex* eines Λ_{nd}-Ausdrucks t wird mittels einer Menge von Regeln definiert, die die Markierungen E (*evaluation*) oder H (*halt*) bis zu einem finalen Teilausdruck in t auf- und abbewegen. Der Algorithmus beginnt mit dem unmarkierten Ausdruck t^E:

1. Erkennung eines NO-Redex:

(ndl/r)	$R[$ `choice` $s\ t\ ]$
(abs)	$R[\ c_A\ v_1 \dots v_{i-1} s_i \dots s_n\ ]$
$(lbeta)$	$R[\ (\backslash x.s)\ t\ ]$
$(lapp)$	$R[\ ($`let` x `=` s `in` $t_x)\ t\ ]$
(cp)	$R[$ `let` x `=` s `in` $R'[x]\ ]$
	falls $s = (\backslash v\,.t)$ oder $s = (c_{A,i}\ v_1 \dots v_n)$
$(llet)$	$R[$ `let` x `= (let` y `=` s `in` $t_y)$ `in` $R'[x]\ ]$
$(lcase)$	$R[$ `case`$_A$ `(let` x `=` s `in` $t_x)$ `of` $alts\ ]$
$(case)$	$R[$ `case`$_A$ $(c_{A,i}\ v_1 \dots v_n)$ `of` $alts\ ]$

2. Erkennung eines irreduziblen Ausdrucks:

   (a) $C[\ (c\ s_1 \dots s_n)^H\ ]$, wenn $n\ >\ ar(c)$.

   (b) $C[\ (\texttt{case}_A\ s\ \texttt{of}\ t_1\texttt{,}\ \dots\texttt{,}t_n)^H\ ]$, wenn $n\ <\ |A|$

   (c) $C[\ (\texttt{case}_A\ s\ \texttt{of}\ t_1\texttt{,}\ \dots\texttt{,}t_{|A|})^H\ ]$, falls s entweder der Form

      - $\lambda x\,.r$ oder
      - $(c_B\ r_1 \dots r_n)$ oder
      - $(c_A\ r_1 \dots r_n)$ mit $n\ \neq\ ar(c)$ entspricht.

3. Erkennung einer WHNF:

   (a) $C[\ (\lambda x\,.r)^E\ ]$. Markiere s als WHNF.

   (b) $C[\ (c\ v_1 \dots v_n)^E\ ]$ und $n\ \leq\ ar(c)$. Markiere als CWHNF.

   (c) $C[\ x^E\ ]$ und x ist eine *freie Variable*. Markiere als VWHNF.[11]

4. Fortsetzung des Markierungsverfahrens, falls keine der obigen Regeln angewandt werden kann:

   (a) $R[\ (s\ t)^E\ ]$. Markiere weiter mit $R[\ (s^E\ t)^e\ ]$.

   (b) $R[\ (\texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ t)^E\ ]$. Markiere weiter in $R[\ (\texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ t^E)^e\ ]$.

   (c) $R[\ \texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ D[x^E]\ ]$ und x ist eine VWHNF.
   Markiere weiter in $R[\ \texttt{let}\ x\ \texttt{=}\ s^E\ \texttt{in}\ D[x^e]\ ]$.

   (d) $R[\ (\texttt{case}_A\ s\ \texttt{of}\ t_1\texttt{,}\ \dots\texttt{,}t_{|A|})^E\ ]$. Markiere weiter in $R[\ (\texttt{case}_A\ s^E\ \texttt{of}\ t_1\texttt{,}\ \dots\texttt{,}t_{|A|})^e\ ]$.

   Das Markierungsverfahren stoppt, falls entweder

   (a) ein NO-Redex oder

   (b) ein irreduzibler Ausdruck oder

   (c) eine WHNF, CWHNF oder VWHNF

   entdeckt wird.

[11] Freie Variablen, die Argumente eines Konstruktors sind, werden nicht als Kopierziel markiert.

Eine *NO-Reduktion* erfolgt, indem der als *NO-Redex* markierte Ausdruck reduziert wird. Der Begriff *NO-Reduktion* wird auch für eine Folge von *NO-Reduktionen* verwendet.

Der Markierungsalgorithmus ist in vier Phasen aufgeteilt: (1) Erkennung eines NO-Redex, (2) Erkennung eines irreduziblen Ausdrucks, (3) Erkennung einer WHNF und (4) der Fortsetzung des Markierungsverfahrens auf einen Teilausdruck. Dies wird in Haskell durch *pattern matching* realisiert. Wenn der überprüfte Teilausdruck mit einem bestimmten Muster übereinstimmt, so wird er entsprechend markiert und der Algorithmus stoppt. Stimmt er nicht überein, so wird er mit dem nächsten Muster verglichen. Folgen keine weiteren Vergleichsmuster, so wird das Haskell-Programm mit einem Fehler abbrechen. Die Implementierung des Algorithmus ist in Anhang A.2 zu finden.

Lemma 3.3.1 Der Markierungsalgorithmus hat folgende Eigenschaften:

- Das Markierungsverfahren terminiert und liefert entweder einen Ausdruck mit eindeutig markierten NO-Redex, oder es markiert den Ausdruck als WHNF, CWHNF oder VWHNF.
- Jeder Super-Ausdruck eines NO-Redex enthält eine Markierung e, die den enthaltenen Redex charakterisiert.
- Geschlossene Ausdrücke können nicht als VWHNF markiert werden.

Der Algorithmus zur Markierung eines NO-Redex im Kalkül $\lambda_{nd,rec}$ unterscheidet sich nur im ersten und dritten Teil.

Definition 3.3.2 (MarkNO$_{rec}$) Der *NO-Redex* eines $\Lambda_{nd,rec}$-Ausdrucks t wird mittels einer Menge von Regeln definiert, die die Markierungen E (*evaluation*) oder H (*halt*) bis zu einem finalen Teilausdruck in t auf- und abbewegen. Der Algorithmus beginnt mit dem unmarkierten Ausdruck t^E:

1. Erkennung eines NO-Redex:

(*ndl/r*)	$R[\ \texttt{choice}\ s\ t\ ]$
(*abs*)	$R[\ c_A\ v_1 \dots v_{i-1} s_i \dots s_n\ ]$
(*lbeta*)	$R[\ (\backslash x.s)\ t\ ]$
(*lapp*)	$R[\ (\texttt{letrec}\ \dots)\ t\ ]$
(*cp*)	$R[\ \texttt{letrec}\ \dots, x_i = s_i, \dots\ \texttt{in}\ R'[x_i]\ ]$ falls $s_i = (\backslash v\,.t)$ oder $s_i = (c_{A,i}\ v_1 \dots v_n)$
(*cp*)	$R[\ \texttt{letrec}\ x_1 = s_1, x_2 = R_2[x_1], \dots, x_j = R_j[x_{j-1}], \dots\ \texttt{in}\ R_\infty[x_j]\ ]$ falls $s_1 = (\backslash v\,.t)$ oder $s_1 = (c_{A,i}\ v_1 \dots v_n)$
(*lletr1*)	$R[\ \texttt{letrec}\ \dots\ \texttt{in}\ (\texttt{letrec}\ \dots\ \texttt{in}\ t)\ ]$
(*lletr2*)	$R[\ \texttt{letrec}\ x_1 = (\texttt{letrec}\ \dots), x_2 = R_2[x_1], \dots, x_j = R_j[x_{j-1}], \dots\ \texttt{in}\ R_\infty[x_j]\ ]$
(*lcase*)	$R[\ \texttt{case}_A\ (\texttt{letrec}\ \dots)\ \texttt{of}\ alts\ ]$
(*case*)	$R[\ \texttt{case}_A\ (c_{A,i}\ v_1 \dots v_n)\ \texttt{of}\ alts\ ]$

2. siehe Punkt (2) in Definition 3.3.1

3. Fortsetzung des Markierungsverfahrens, falls keine der obigen Regeln angewandt werden kann:
   (a) $R[\ (s\ t)^E\ ]$. Markiere weiter in $C[\ (s^E\ t)^e\ ]$.
   (b) $R[\ (\texttt{letrec}\ \ldots\ \texttt{in}\ t)^E\ ]$. Markiere weiter in $R[\ (\texttt{let}\ \ldots\ \texttt{in}\ t^E)^e\ ]$.
   (c) $R[\ \texttt{letrec}\ x_1 = s_1, x_2 = R_2[x_1], \ldots, x_j = R_j[x_{j-1}], \ldots\ \texttt{in}\ R_\infty[x_j^E]\ ]\ ]$ und x ist eine VWHNF. Markiere weiter in
   $R[\ \texttt{letrec}\ x_1 = s_1^E, x_2 = R_2[x_1], \ldots, x_j = R_j[x_{j-1}], \ldots\ \texttt{in}\ R_\infty[x_j^e]\ ]\ ]$.
   (d) $C[\ (\texttt{case}_A\ s\ \texttt{of}\ t_1, \ldots, t_{|A|})^E\ ]$. Markiere weiter in $R[\ (\texttt{case}_A\ s^E\ \texttt{of}\ t_1, \ldots, t_{|A|})^e\ ]$.
4. siehe Punkt (4) in Definition 3.3.1

3.3.2 Implementierung des Markierungsverfahrens

Die Markierung eines Redex in Normalordnung für beide Kalküle ist in einer Funktion zusammengefaßt. Der Markierungsalgorithmus sucht innerhalb des Ausdrucks einen geeigneten Teilterm und markiert diesen. Die Markierung des Subausdrucks muß während des rekursiven Abarbeitens dem Superausdruck „nach oben“ weitergegeben werden. Aus diesem Grund benutzen wir den Datentyp `MarkedCoreExpr`, der den Ausdruck näher charakterisiert.

```
data MarkedCoreExpr
     = HasRedex ReductionType         CoreExpr
     | HasWHNF                        CoreExpr
     | HasVWHNF VName                 CoreExpr
     | HasCWHNF (CName,CArity) CArity CoreExpr
     | HasError String                CoreExpr
     deriving (Eq, Show)
```

Enthält der Ausdruck t einen Redex, dann wird der Reduktionstyp und der Ausdruck selbst an den Konstruktor `HasRedex` gebunden. Ein Ausdruck in WHNF wird durch `HasWHNF` gekennzeichnet. Zur Charakterisierung einer VWHNF wird noch der Name der Variablen mitgeführt. Enthält der Ausdruck eine Konstruktoranwendung, dann werden der Typ und die Stelligkeit vermerkt, sowie die maximale Anzahl von Argumenten, auf die die Konstruktoranwendung noch angewendet werden kann. Dadurch können übersättigte Konstruktoranwendungen erkannt werden. Solch ein Fehler im Ausdruck wird mittels des Konstruktors `HasError` an den Superausdruck übergeben.

Die Markierungsfunktion `markNORedex` kann teilweise direkt aus den Definitionen 3.3.1 und 3.3.2 in Haskell-Code übersetzt werden.

Eine Variable wird als VWHNF, eine Abstraktion als WHNF und eine Konstruktoranwendung als CWHNF markiert:

```
markNORedex rec expr@(EVar _ v)
    = HasVWHNF v (EVar VWHNF v)

markNORedex rec expr@(ELam _ v e)
    = HasWHNF (ELam WHNF v e)

markNORedex rec expr@(ECons _ t a)
    = HasCWHNF (t,a) a expr
```

Innerhalb eines `choice`-Ausdrucks wird nicht weiter markiert, weshalb wir direkt den (nd)-Redex markieren können.

```
markNORedex rec expr@(EChoice _ e1 e2)
    = HasRedex ND (EChoice (Redex ND) e1 e2)
```

Bei Applikationen gibt es mehrere Möglichkeiten, den Ausdruck zu markieren. Wird ein `let(rec)`-Ausdruck oder eine Abstraktion auf einen Ausdruck angewendet, dann wird die Applikation als (*lapp*)- bzw. (*lbeta*)-Reduktion markiert.

```
markNORedex rec expr@(EApp _ e1@(ELam _ _ _) e2)
    = HasRedex LBeta (EApp (Redex LBeta) e1 e2)

markNORedex rec expr@(EApp _ e1@(ELet _ _ _ _) e2)
    = HasRedex LApp (EApp (Redex LApp) e1 e2)
```

Ansonsten kann der linke Ausdruck der Applikation nur noch einen weiteren Redex, der auch als nächstes reduziert werden muß, eine VWHNF oder eine Konstruktoranwendung enthalten. Deshalb wird erst geprüft, ob es sich um eine Konstruktoranwendung handelt. In diesem Fall wird die Sättigung des Konstruktors überprüft, und falls alle Argumente Variablen sind, der Ausdruck als CWHNF markiert. Wenn eines der Argumente keine Variable ist, dann wird der Ausdruck als (*abs*)-Redex markiert. Die Argumente werden von links nach rechts betrachtet, wie es die Definition von (*abs*) vorschreibt.

Falls ein Ausdruck eine Konstruktoranwendung ist, dann liefert `getTagArityArgs` die eindeutige Konstruktorbezeichnung (*tag*), die Stelligkeit (*arity*) und eine Liste der Argumente des Konstruktors (`as`).

```
markNORedex rec expr@(EApp m e1 e2)
    = case getTagArityArgs expr
        of Just (t,a,as)
             -> if a < length_as
                   then HasError ("Reduction.markNORedex (App): "
                                  ++ "oversaturated constructor ‘"
                                  ++ show expr ++ "‘") expr
                   else if all isVariable as
                           then HasCWHNF (t,a) (a - length_as) expr
                           else HasRedex Abs (EApp (Redex Abs) e1 e2)
                   where
                     length_as = length as
           Nothing
             -> case markNORedex rec e1
                  of HasError    s    e1’ -> HasError s expr
                     HasRedex    r    e1’ -> HasRedex r (EApp m e1’ e2)
                     HasVWHNF    v    e1’ -> HasVWHNF v (EApp m e1’ e2)
```

Ein `case`-Ausdruck kann als erstes Argument einen `let(rec)`-Ausdruck enthalten, so daß der Term als (*lcase*)-Redex markiert werden muß.

```
markNORedex rec (ECase m t expr@(ELet _ _ _ _ ) es)
    = HasRedex CaseLet (ECase (Redex CaseLet) t expr es)
```

Ansonsten wird das erste Argument mittels `markNORedex` rekursiv markiert. Dabei müssen alle Ergebnismöglichkeiten überprüft werden. Ein enthaltener Redex (`HasRedex`), ein Fehler (`HasError`) und eine VWHNF (`HasVWHNF`) werden an den Super-Term „durchgereicht“, eine WHNF (`HasWHNF`) und eine ungesättigte, nicht-pure Konstruktoranwendung (`HasCWHNF`) liefern einen Fehler, da diese nicht reduziert werden können.

```
markNORedex rec expr@(ECase m t e as)
    = case markNORedex rec e
        of HasError s’       e’ -> HasError s’ expr
           HasRedex r’       e’ -> HasRedex r’ (ECase m t e’ as)
           HasWHNF           e’ -> HasError ("Reduction.markNORedex (Case): "
                                             ++ "unreducable expression: "
                                             ++ show expr) expr
           HasVWHNF v’       e’ -> HasVWHNF v’ (ECase m t e’ as)
           HasCWHNF (t’,_) 0 e’ -> if 1 == length [n
                                                  | (n, _, _) <- as
                                                  , n == t’
                                                  ]
                                        then HasRedex Case (ECase (Redex Case)
                                                  t e as)
                                        else HasError ("Reduction.markNORedex."
                                                   ++ "Case: cannot match ‘"
                                                   ++ show e ++ "‘ in ‘"
                                                   ++ show expr ++ "‘")
                                                  expr
           HasCWHNF _      _ _  -> HasError ("Reduction.markNORedex (Case): "
                                             ++ "unsaturated constructor in "
                                             ++ "case expression: "
                                             ++ show expr) expr
```

Im nicht-rekursiven Kalkül kann sich der NO-Redex im Term `e` befinden, wobei dieser Redex dem Superausdruck zugewiesen wird. Ist `e` in WHNF, dann auch der umgebende `let`-Ausdruck. Gleiches gilt für den CWHNF-Fall. Enthält `e` jedoch eine VWHNF, dann muß geprüft werden, ob die Variable in VWHNF die gebundene Variable des `let`-Ausdrucks ist. In diesem Fall (1) könnte der an `v` gebundene Ausdruck `tv` kopiert werden, wenn er eine Abstraktion oder eine pure Konstruktoranwendung ist.

```
markNORedex False expr@(ELet m rec bs@[(v,tv)] e)
    = case markNORedex False e
        of HasError   s  e' -> HasError s  expr
           HasRedex   r  e' -> HasRedex r  (ELet m    False bs e')
           HasWHNF       e' -> HasWHNF     (ELet WHNF False bs e)
           HasCWHNF ta a e' -> HasCWHNF ta a expr
           HasVWHNF v'   e' -> if v /= v'
                                  then HasVWHNF v' (ELet m False bs e')
                                  else markBinding v' e' -- (1)
        where
```

Der Bindungsausdruck `tv` ist ohne weiteres kopierbar, wenn er eine Abstraktion ist. Sollte `tv` ein `let`-Ausdruck sein, dann muß der gesamte Ausdruck als (*llet*)-Redex markiert werden. Enthält `tv` einen Redex (`markBoundExpr`), dann wird dieser zur Markierung des umgebenden `let` benutzt.

```
      markBinding v' e'
        = case tv
            of ELam _ _ _   -> copyRedex e'
               ELet _ _ _ _ -> HasRedex LLet (ELet (Redex LLet) False bs e)
               otherwise    -> markBoundExpr e'
```

Mit `markBoundExpr` wird versucht, weiter innerhalb des Terms `tv` zu markieren. Dabei müssen Fehler abgefangen, Redexe markiert und VWHNFs erkannt werden. Ansonsten kann `tv` nur noch eine kopierbare Konstruktoranwendung sein, so daß der gesamte Ausdruck als (*cp*)-Redex markiert werden kann.

```
      markBoundExpr e'
        = case markNORedex False tv
            of HasError s   tv' -> HasError s   expr
               HasRedex r'  tv' -> HasRedex r'  (ELet m False [(v,tv')] e)
               HasVWHNF v'' tv' -> HasVWHNF v'' (ELet m False [(v,tv')] e)
               otherwise        -> copyRedex e'

      copyRedex e'
        = HasRedex (Copy v) (ELet (Redex (Copy v)) False bs e')
```

Im rekursiven Kalkül müssen wir mehrere Fallunterscheidungen bei `letrec`-Ausdrücken beachten. Im Gegensatz zum nicht-rekursiven Kalkül gibt es zwei (*llet*)-Varianten.

Die erste Variante (*lletr1*) faßt hintereinander geschaltete `letrec`-Ausdrücke zu einem einzigen zusammen, da im rekursiven Kalkül die Reihenfolge der Bindungen nicht wichtig ist[12].

[12] Wegen unserer Konvention, daß alle Variablennamen unterschiedlich sind, ist dies ohne weiteres möglich.

```
markNORedex True expr@(ELet _ _ bs e@(ELet _ _ _ _))
    = HasRedex LLetrec1 (ELet (Redex LLetrec1) True bs e)
```

Die zweite Variante (*lletr2*) ist wie das nicht-rekursive (*llet*) definiert, jedoch muß im Falle einer VWNHF im Term `e` geprüft werden, ob die markierte Variable eine der `letrec`-gebundenen Variablen ist.

Wegen des rekursiven Verfolgens von Termen in VWNHF ist die Markierung eines (*lletr2*)-Redex etwas komplizierter als im nicht-rekursiven Kalkül.

```
markNORedex True expr@(ELet m True bs e)
    = case markNORedex True e
        of HasError s    e' -> HasError s  expr
           HasRedex r    e' -> HasRedex r  (ELet m    True bs e')
           HasWHNF       e' -> HasWHNF     (ELet WHNF True bs e )
           HasCWHNF ta a e' -> HasCWHNF ta a expr
           HasVWHNF v    e' -> if v `notElem` boundVars
                                  then HasVWHNF v (ELet m True bs e')
                                  else checkBindings v v "" [v] e'
        where
```

Der Markierungsalgorithmus muß nun alle Bindungen derart untersuchen, daß bei jedem Auftreten einer VWHNF geprüft wird, ob die Variable als Bindung des `letrec`-Ausdrucks deklariert ist. Die Bindungen werden so lange durchsucht bis alle Variablen-Bindungen maximal genau einmal betrachtet wurden. In diesem Fall enthält der betrachtete `letrec`-Ausdruck eine unauflösliche Schleife von Referenzen.

```
boundVars
  = [x | (x,_) <- bs]

hasRedex red mark bs' e'
  = HasRedex red (ELet mark True bs' e')

newbs var e'
  = updateBinding var e' bs

cycleError
  = HasError ("Reduction.markNORedex (Letrec): "
             ++ "cyclic term in expression: "
             ++ show expr) expr
```

Rekursives Überprüfen der einzelnen Bindungen:

```
checkBindings copyVar refVar tVar visVars e'
  = case vexpr
      of ELam _ _ _   -> copyRedex
         ELet _ _ _ _ -> lletrec2Redex
         otherwise    -> checkBoundExprs
      where
```

```
vexpr
  = lookupBinding refVar bs

copyRedex
  = if copyVar == refVar
       then hasRedex (Copy copyVar) (Redex (Copy copyVar)) bs e'
       else hasRedex (Copy refVar ) (Redex (Copy refVar ))
                                     (newbs tVar e') e

lletrec2Redex
  = hasRedex (LLetrec2 refVar) (Redex (LLetrec2 refVar)) bs e
```

Der aktuelle Bindungsterm wird markiert. Bei Fehlern und Redex-Markierungen wird der Super-Term entsprechend markiert. Wird jedoch eine VWHNF gefunden, dann müssen die restlichen Bindungen untersucht werden (`checkCycles`).

```
checkBoundExprs
  = case markNORedex True vexpr
      of HasError s  ve -> HasError s  e
         HasRedex r' ve -> hasRedex r' m (newbs refVar ve) e
         HasCWHNF _ _ _ -> copyRedex
         HasVWHNF v' ve -> checkCycles v' ve
```

Ist die aktuelle Variable schon einmal überprüft worden, dann kann der Ausdruck nicht markiert werden, da er eine unauflösliche Schleife enthält.

```
checkCycles v' ve
  = if v' `elem` visVars
       then cycleError
       else if v' `elem` boundVars
               then checkBindings copyVar v' refVar (v':visVars) ve
               else HasVWHNF v' (ELet m True (newbs refVar ve) e)
```

3.3.3 Markierung interner Redexe

Zur Validierung der Reduktionsdiagramme benötigen wir neben den NO-Redexen auch diejenigen Redexe, die nicht die Auswertung in Normalordnung bestimmen. Diese nennen wir im folgenden *interne Redexe*.

Definition 3.3.3 Die Menge $\mathcal{R}(M)$ ist die *Menge aller Redexe* des Ausdrucks M. Die Menge $\mathcal{R}_{NO}(M)$ ist die *Menge aller Redexe bei Auswertung in Normalordnung*, wobei wegen der Eindeutigkeit des auszuwertenden Redex $|\mathcal{R}_{NO}(M)| \leq 1$ gilt. Die *Menge aller internen Redexe* $\mathcal{R}_I(M)$ enthält alle Redexe eines Reduktionskontextes, die kein NO-Redex sind.

Die Funktion `markReduction` markiert im Ausdruck `e` entsprechend des Kalküls (für die rekursive Version besitzt `rec` den Wert `True`) einen NO-Redex vom Typ `r`. Da im NO-Markierungsalgorithmus der `choice`-Konstrukt mit (nd) markiert wird, müssen in der Funktion `markReduction` noch die Fälle für (ndl) und (ndr) überprüft und die Markierung entsprechend geändert werden. Jeder andere Redex-Typ wird mit dem geforderten Typ verglichen.

Einige Reduktionstypen, wie z. B. (*cp*), besitzen noch spezielle Typen, im Falle (*cp*) sind dies (*cpd*) und (*cpt*). Dabei definiert die Funktion `isSameReductionType` eine Gleichheit über Reduktionstypen. Z. B. ist (*cpd*) eine (*cp*)-Reduktion, jedoch nicht umgekehrt. In der Implementierung wird nicht die Äquivalenzklasse `Eq` benutzt, weil intern jede kopierende Reduktion die Zielvariablen enthält. Diese Variable muß natürlich in bestimmten Fällen mitverglichen werden. Im Falle des Typ-Vergleichs ist die Variable aber unwichtig.

```
markReduction rec (NO r) e
    = case markNORedex rec e
         of HasRedex ND e' -> case r
                                of ND        -> [e']
                                   NDL       -> [replaceND NDL e']
                                   NDR       -> [replaceND NDR e']
                                   otherwise -> []
           HasRedex r' e' -> if r `isSameReductionType` r'
                               then [e']
                               else []
           otherwise      -> []
```

Der Algorithmus zur Bestimmung von Redexen (`markRedex`) liefert als Ergebnis eine Menge von markierten Ausdrücken. Einer dieser Redexe kann natürlich ein NO-Redex sein, wenn der Ausdruck noch weiter in Normalordnung reduziert werden kann. Um diesen Redex aus der Menge von Redexen herauszunehmen, muß er mittels des NO-Markierungsalgorithmus gefunden werden, falls er existiert. Der gefundene NO-Redex wird dann aus der Menge gefiltert. Da in diesem Fall nicht nur der Redex-Typ sondern der gesamte Ausdruck inkl. seiner Markierung(en) verglichen wird, benötigen wir eine Instanz der `Eq`-Klasse für Reduktionstypen (siehe Anhang 142).

```
markReduction rec (I r) e
    = let es = markRedex rec r e
       in case markNORedex rec e
            of HasRedex r' e' -> [e'' | e'' <- es, e'' /= e']
               otherwise      -> es
```

Wird der Reduktionstyp als allgemein gekennzeichnet, dann wird ein möglicher NO-Redex nicht ausgefiltert.

```
markReduction rec (R r) e
    = markRedex rec r e
```

Der Mengentyp, den wir zur Darstellung allgemeiner Reduktionsdiagramme verwenden, kann natürlich nicht markiert werden, so daß eine Fehlermeldung ausgegeben wird.

```
markReduction rec (SET _ _ _) e
    = error "Cannot mark 'set construct'!"
```

Für das Hinzufügen von Reduktionsarten zum Basis-Kalkül ist es nötig, die Markierungsfunktion für die neuen Reduktionen entsprechend zu erweitern. Dazu benutzen wir als Rahmenfunktion `markRedex'`, der entsprechend (einfache) Markierungsfunktionen als Parameter übergeben werden können. Dadurch ist es möglich, durch kleine Änderungen des Quellcodes den zugrunde liegenden Reduktionsmechanismus zu erweitern. Die Parameter `lam` und `cons` erlauben eine Markierung innerhalb von Abstraktionen bzw. Konstruktoranwendungen, `rec` bestimmt, ob der rekursive Kalkül benutzt werden soll, und `redexType` ist der zu wählende Reduktionstyp.

```
markRedex' s@(lam,cons) rec redexType
    = case redexType
        of ----------- Basisregeln ------------
           LBeta      -> markContextR markLBeta
           LApp       -> markContextR markLApp
           LLet       -> markContextR (markLLet rec)
           LLetrec1   -> markContextR markLLetrec1
           LLetrec2 v -> markContextR markLLetrec2
           Abs        -> markContextS markAbs
           Copy     v -> markContextR markCopy
           CopyT    v -> markContextR markCopyT
           CopyD    v -> markContextR markCopyD
           Case       -> markContextR markCase
           CaseLet    -> markContextR markCaseLet
           ND         -> markContextR markND
           NDL        -> markContextR markNDL
           NDR        -> markContextR markNDR

           -------- Erweiterungsregeln --------
           LDel     v -> markContextR markLDel
           LDelS    v -> markContextS markLDel
           LDelCyc1 v -> markContextR markLDelCyc1
           LDelCyc2   -> markContextR markLDelCyc2
           LDup     v -> markContextR markLDup
           LCVT     v -> markContextR markLCVT
           LCVD     v -> markContextR markLCVD
           LCom       -> markContextR markLCom
           UCP      v -> markContextR markUCP
           EQL      p -> markContextC markEQL (True,True)
```

Als *high level*-Funktion, die nicht innerhalb von Abstraktionen und Konstruktoranwendungen markiert, wird die Funktion `markRedex` verwendet.

```
markRedex = markRedex' (False,False)
```

Jeder Reduktionstyp kann in einem bestimmten Kontext gesucht werden (siehe z. B. die nicht-deterministischen Reduktionstypen (nd), (ndl) und (ndr)). `markContextR` markiert Redexe der Klasse R (siehe Definitionen 3.1.3 und 3.1.4).

Markierung in C-Kontexten

Das Loch eines C-Kontextes kann an jeder Stelle des Ausdrucks sein. Mittels des Schalters `s` muß jedoch explizit angegeben werden, ob die Markierungsfunktion `marker` auch innerhalb von Abstraktionen und Konstruktoranwendungen angewandt werden soll. Dadurch kann eine Funktion zur Kontext-Markierung mehrfach genutzt werden (siehe Oberflächenkontext). Wir werden nämlich bei bestimmten Reduktionen Einschränkungen bzgl. der zu betrachteten Kontexte machen müssen.

`markContextC` ist rekursiv definiert. Der aktuelle Knoten des Ausdrucks wird markiert (`marker expr`). Das Ergebnis dieser Markierung ist eine Liste von Termen, die alle verschiedene Markierungen besitzen. Innerhalb von Abstraktionen wird nur markiert, wenn der Schalter `inLambda` gleich `True` ist, wobei rekursiv alle möglichen, markierten Terme gesucht werden.

```
markContextC marker s@(inLambda,_) expr@(ELam _ v e)
    =    marker expr
      ++ [ ELam Unmarked v e'
         | inLambda
         , e' <- markContextC marker s e
         ]
```

Bei Applikationen wird normalerweise in beiden Zweigen des Ausdrucks (`e1`, `e2`) weiter markiert. Ist `inConstructor` gleich `False`, dann muß geprüft werden, ob Term `e1` eine (ungesättigte) Konstruktoranwendung ist, da in diesem Fall nicht innerhalb von `e2` markiert werden darf.

```
markContextC marker s@(_,inConstructor) expr@(EApp _ e1 e2)
    =    marker expr
      ++ [ EApp Unmarked e1' e2
         | e1' <- markContextC marker s e1
         ]
      ++ [ EApp Unmarked e1 e2'
         | inConstructor || not (isConstructor e1)
         , e2' <- markContextC marker s e2
         ]
```

Bei `let(rec)`-Ausdrücken wird sowohl in den Bindungen als auch im Term `e` markiert, wobei im Falle des rekursiven Kalküls in allen Bindungen markiert werden muß.

```
markContextC marker s expr@(ELet _ rec bs e)
    =    marker expr
      ++ [ ELet Unmarked rec bs e'
         | e' <- markContextC marker s e
         ]
      ++ [ ELet Unmarked rec bs' e
         | (v,be) <- bs
         , be' <- markContextC marker s be
         , let bs' = updateBinding v be' bs
         ]
```

`case`-Ausdrücke können ebenfalls Redexe im Term `e` und in jeder Alternative enthalten.

```
markContextC marker s expr@(ECase _ tp e as)
    =    marker expr
      ++ [ ECase Unmarked tp e' as
         | e' <- markContextC marker s e
         ]
      ++ [ ECase Unmarked tp e  as'
         | (t,vs,ae) <- as
         , ae' <- markContextC marker s ae
         , let as' = updateAltExpr t ae' as
         ]
```

Beim `choice`-Konstrukt werden wie bei Applikationen beide Teilausdrücke durchsucht.

```
markContextC marker s expr@(EChoice _ e1 e2)
    =    marker expr
      ++ [ EChoice Unmarked e1' e2
         | e1' <- markContextC marker s e1
         ]
      ++ [ EChoice Unmarked e1  e2'
         | e2' <- markContextC marker s e2
         ]
```

Abschließend können nur noch Terme ohne Teilausdrücke, also Variablen und Konstruktoren, markiert werden.

```
markContextC marker s expr
    = marker expr
```

Markierung in Reduktionskontexten

Die in den Definitionen 3.1.3 und 3.1.4 beschriebenen Kontexte werden mittels der Funktion `markContextR` implementiert.

Im Gegensatz zur Markierung in C-Kontexten sind hier weniger Definitionen nötig, da nur unterhalb bestimmter Sprachkonstrukte weiter markiert wird. Bei `let`(`rec`)-Ausdrücken wird normalerweise nur unterhalb von Term `e` weiter reduziert. Wenn jedoch die gebundene Variable in den Term `e` kopiert werden kann (`e` enthält eine VWHNF für `v`), dann muß auch der an `v` gebundene Term untersucht werden. Im Falle von `letrec` gilt das natürlich für alle Bindungen.

```
markContextR marker expr@(ELet _ rec bs e)
    =    marker expr
      ++ [ ELet Unmarked rec bs e'
         | e' <- markE marker e
         ]
      ++ [ ELet Unmarked rec bs' e
         | (v,be) <- bs
         , [] /= markE (markVar v) e
         , be' <- markContextR' marker be
         , let bs' = updateBinding v be' bs
         ]
      where
        markE = if rec then markContextR' else markContextR
```

```
markContextR marker expr
    = markContextR' marker expr
```

`markContextR'` markiert Kontexte der Klasse R^-. Es ist leicht zu sehen, daß das Markierungsverfahren nur Applikationen und `case`-Ausdrücke betrachtet.

```
markContextR' marker expr@(EApp _ e1 e2)
    =    marker expr
      ++ [ EApp Unmarked e1' e2
         | e1' <- markContextR' marker e1
         ]

markContextR' marker expr@(ECase _ tp e as)
    =    marker expr
      ++ [ ECase Unmarked tp e' as
         | e' <- markContextR' marker e
         ]

markContextR' marker expr
    = marker expr
```

Markierung in Oberflächenkontexten

Die in Definition 3.1.5 beschriebenen Kontexte werden mittels der Funktion `markContextS` markiert. Da Oberflächenkontexte und C-Kontexte sich nur dadurch unterscheiden, daß der Redex nicht innerhalb einer Abstraktion vorkommen darf, gestaltet sich die Definition ganz einfach. Wir verwenden die Funktion `markContextC`, wobei der Parameter `lam` auf `False` gesetzt wird:

```
markContextS marker
    = markContextC marker (False,True)
```

Die Markierungsfunktion `marker`

Den Kontextmarkierungen `markContextC`, `markContextR` oder `markContextS` können als Markierungsfunktion `marker` beispielsweise die Funktion `markLetApp` übergeben werden:

```
markLApp (EApp _ e1@(ELet _ _ _ _) e2) = [EApp (Redex LApp) e1 e2]
markLApp expr                          = []
```

Sie markiert in einem Ausdruck Applikationen, deren linker Term ein `let`-Ausdruck ist. `(markRedex False LApp)` liefert dann die Menge aller Ausdrücke zurück, die *genau eine* Redex-Markierung (NO- oder I-Redex) vom Typ (*lapp*) enthalten.

Eine etwas kompliziertere Markierungsfunktion ist `markCopy'`, die einen (*cp*)-Redex bzw. (*cpt*)- oder (*cpd*) (siehe Kapitel 7.2.6) markiert. Wir verwenden diese Funktion als Rahmenfunktion für die beiden Teilreduktionen (*cpt*) und (*cpd*) sowie für (*cp*) selbst. Dazu werden die einzelnen Markierungsfunktionen über `markCopy'` definiert, wobei eine entsprechende Kennung übergeben wird, die innerhalb von `markCopy'` den richtigen Kontext und die Markierung bestimmt.

```
markCopy  = markCopy' 1

markCopyT = markCopy' 2

markCopyD = markCopy' 3
```

Für alle kopierbaren Variablenbindungen (pure Konstruktoranwendung oder Abstraktionen) werden im Term `expr` — im rekursiven Kalkül auch innerhalb der Bindungen `bs` — alle Kopierziele markiert. An diesem Beispiel ist auch deutlich zu sehen, daß innerhalb der Markierungsfunktion wiederum andere Redexe (in diesem Fall Variablen) sogar in verschiedenartigen Kontexten markiert werden können.

```
markCopy kind (ELet _ rec bs expr)
    = [ ELet (Redex (copyredex var)) rec bs expr'
      | (var,tvar) <- cpvars
      , expr'      <- markcontext (markVar tvar) expr
      ]
      ++
      [ ELet (Redex (copyredex var)) rec bs' expr
      | rec
      , (var,tvar) <- cpvars
      , (v,e')     <- bs
      , v /= var
      , e          <- markcontext (markVar tvar) e'
      , let bs'     = updateBinding v e bs
      ]
      where
```

Die an Abstraktionen oder puren Konstruktoranwendungen gebundenen Variablen werden in der Liste von Variablenpaaren gesammelt (`cpvars`). Diese auf den ersten Blick ungewöhnliche Vorgehensweise rührt daher, daß später bei (cp)-Reduktionen Indirektionen berücksichtigt werden sollen, wobei die erste Komponente des Paares das Kopierziel beschreibt und die zweite die Variable des zu kopierenden Terms.

```
        cpvars
          = concat [ [(v,v)]
                   | (v,e) <- bs
                   , isPureConstructor e || isLambda e
                   ]
```

Mittels des Parameters `copy` wird die Art der Reduktion spezifiziert. Wir numerieren einfach alle möglichen Redex-Arten durch, um Erweiterungen so einfach wie möglich zu halten. Entsprechend wird die Markierung der Kopierziele implementiert.

```
        markcontext f e
          = case kind
              of 1 -> markContextC f (True,True) e
                 2 -> markContextS f e
                 3 -> markContextC f (True,True) e
                      \\
                      markContextS f e
```

Auf dieselbe Weise wird der eigentliche Redex markiert.

```
        copyredex v
          = case kind
              of 1 -> Copy  v
                 2 -> CopyT v
                 3 -> CopyD v
```

Alle anderen Terme besitzen keinen (cp)-Redex, so daß die Ergebnismenge leer ist.

```
markCopy kind expr = []
```

3.4 Implementierung des Reduktionsmechanismus

Ist ein Redex in einem Ausdruck markiert, dann kann dieser reduziert werden. Dazu definieren wir die Funktion `reduceCoreExpr`, die den markierten Ausdruck rekursiv durchläuft und an der markierten Stelle den ensprechenden Reduktionsschritt durchführt.

Der Reduziermechanismus ist derart aufgebaut, daß ein markierter Ausdruck entsprechend reduziert wird. Das Ergebnis ist eine Menge von Ausdrücken, wobei alle NO-Reduktionen genau ein Redukt besitzen, außer (nd)- und interne Reduktionen.

(*lbeta*)

Die $(lbeta)$-Reduktion erzeugt aus einem λ-Ausdruck einen neuen `let(rec)`-Term.

```
reduceCoreExpr rec level (EApp (Redex LBeta) (ELam _ v e) e')
    = [ELet Unmarked rec [(v,e')] e]
```

(*lapp*)

Die $(lapp)$-Reduktion schiebt ein `let(rec)` aus einer Applikation heraus.

```
reduceCoreExpr rec level (EApp (Redex LApp) (ELet _ _ bs e) e')
    = [ELet Unmarked rec bs (EApp Unmarked e e')]
```

Ein als (abs) markierter Redex wird zu einem `let(rec)`-Ausdruck mit neu eingeführter Variable reduziert. Neu eingeführte Variablen werden durch einen eindeutigen Präfix[13] und der Nummer des aktuellen Reduktionsschritts (`level`) errechnet. Das am weitesten links stehende Argument des Konstruktors[14] wird durch die neue Variable ersetzt, der abstrahierte Ausdruck durch das `let(rec)` an `v` gebunden.

[13] Um zu vermeiden, daß zufällig dieser Präfix als Variablenname bei der Eingabe von Termen gewählt wird, werden diese Fälle vom Parser abgelehnt.

[14] Durch die Markierungsfunktion muß gewährleistet sein, daß die markierte Applikation auch wirklich eine Konstruktoranwendung ist.

```
reduceCoreExpr rec level expr@(EApp (Redex Abs) _ _)
    = [ELet Unmarked rec [(v,be)] expr']
      where
        v = newVarPrefix ++ show level
        (Just be, expr') = exchangeLeftmostExpr v expr
```

(llet)

Im nicht-rekursiven Kalkül ist die (*llet*)-Reduktion einfach, das innere `let` wird „nach außen“ geschoben.

```
reduceCoreExpr False level (ELet (Redex LLet) False [(x,ELet _ _ yb tx)] s)
    = [ELet Unmarked False yb (ELet Unmarked False [(x,tx)] s)]
```

Im rekursiven Kalkül ist die (*llet*)-Reduktion in die zwei Reduktionen (*lletr1*) und (*lletr2*) aufgeteilt, da wir auch gestaffelte `letrec`-Ausdrücke zu einem einzelnen zusammenfassen können.

```
reduceCoreExpr True level (ELet (Redex LLetrec1) True bs1 (ELet _ _ bs2 e))
    = [ELet Unmarked True (bs1 ++ bs2) e]

reduceCoreExpr True level (ELet (Redex (LLetrec2 v)) True bs1 e1)
    = [ELet Unmarked True (bs1' ++ bs2) e1]
      where
        (bs2,e2)
          = case lookupBinding v bs1
                of (ELet _ _ bs e) -> (bs,e)
                   otherwise       -> error $ "Reduction.reduceCoreExpr "
                                           ++ "(lletr2): "
                                           ++ "wrong marked expression."
        bs1'
          = updateBinding v e2 bs1
```

(cp)

Innerhalb der Bindungen und des Terms `e` wird der an `v` gebundene Ausdruck `be` an die als VWHNF markierte Stelle kopiert. Die Funktion `replaceMarkedVar` übernimmt diese Aufgabe, wobei sie auch noch den kopierten Term umbenennt, um unserer Variablen-Konvention zu genügen[15]. Wenn der erste Parameter `True` ist, dann wird auf jeden Fall an die Stelle der VWHNF kopiert, auch wenn die Variablen unterschiedlich sind. Aus diesem Grund darf nur genau eine Variable in `e` als VWHNF markiert sein. Der Nutzen dieser Vorgehensweise ist, daß bei einer Variante der (*cp*)-Reduktion, die Indirektionen der Kopier-Variable erlaubt (siehe Kapitel 7.5, Seite 129), die Variablen unterschiedlich sind und daher auch „fremde“ Variablen im Zielterm ersetzt werden müssen.

[15]Das Umbenennen kann abgeschaltet werden, wenn der zweite Parameter der leere String ist (`""`). Siehe auch Anhang A.1.

```
reduceCoreExpr rec level (ELet (Redex (Copy v)) _ bs e)
    = [ELet Unmarked rec bs' e']
      where
        be  = lookupBinding v bs
        e'  = replaceMarkedVar True (show level) v be e
        bs' = [ (v1,e1')
              | (v1,e1) <- bs
              , let e1' = replaceMarkedVar True (show level) v be e1
              ]
```

(*case*)

Die Reduktion eines `case`-Redex sieht etwas kompliziert aus, da der Konstruktor überprüft und die richtige Alternative gefunden werden muß.

```
reduceCoreExpr rec level e@(ECase (Redex Case) tp con as)
    = case getTagArityArgs con
        of Nothing          -> error $ "Reduction.reduceCoreExpr: ‘"
                                       ++ show con
                                       ++ "‘ is not a constructor application"
           Just (t',a',es') -> if a' /= length es'
                                  then error $ "Reduction.reduceCoreExpr: ‘"
                                               ++ show con
                                               ++ "‘ is not saturated"
                                  else reduceCase (t',a',es')
```

Nach der Überprüfung des Konstruktors kann nach der passenden Alternative gesucht werden. Auch hier kann es zu Fehlern kommen, die abgefangen werden müssen.

```
        where
          reduceCase (t',a',es')
            = case [a | a@(n,vs,e) <- as, n == t']
                of []           -> error $ "Reduction.reduceCoreExpr: "
                                           ++ "Cannot match ‘"
                                           ++ show con ++ "‘ in ‘"
                                           ++ show e ++ "‘"
```

Ist die passende Alternative gefunden, dann kann sie bei nullstelligen Konstruktoren sofort übernommen werden. Ansonsten müssen die *pattern*-Variablen in der Alternative durch neue Variablennamen ersetzt werden, und mit den Variablen der puren Konstruktoranwendung assoziiert werden. Die neu erstellten Bindungen müssen letztlich nur noch dem Kalkül entsprechend als geschachtelte `let`-Ausdrücke oder als ein einzelnes `letrec` zum Alternativenterm hinzugefügt werden.

```
                    (_,avs,ae):_ -> if rec
                                       then case bs
                                              of [] -> [ak]
                                                 _  -> [ELet Unmarked rec bs ak]
                                       else [addLets bs ak]
                                    where
                                      vs = [ newVarPrefix
                                             ++ show level ++ "_" ++ x
                                           | x <- avs
                                           ]
                                      nv = zip avs vs
                                      ak = renameFreeVars nv ae
                                      bs = zip vs es'
```

(lcase)

Das `let(rec)` innerhalb eines `case`-Ausdrucks wird wie bei (*llet*) bzw. (*lletr1*) und (*lapp*) nach außen geschoben.

```
reduceCoreExpr rec level (ECase (Redex CaseLet) tp e@(ELet m' _ bs' e') as)
    = [ELet m' rec bs' (ECase Unmarked tp e' as)]
```

(nd)

Die (*nd*)-Reduktion teilt sich in die zwei Reduktionen (*ndl*) und (*ndr*). Normalerweise werden auch nur diese beiden Reduktionen einzeln betrachtet. Es ist jedoch auch möglich, sie zusammengefaßt als (*nd*) zu verwenden.

```
reduceCoreExpr rec level (EChoice (Redex ND) e1 e2)
    = [e1,e2]

reduceCoreExpr rec level (EChoice (Redex NDL) e1 e2)
    = [e1]

reduceCoreExpr rec level (EChoice (Redex NDR) e1 e2)
    = [e2]
```

Reduktion der Kalkül-Erweiterungen

An dieser Stelle müssen wir einen Schnitt machen, da nun die Implementierung der Reduktionsregeln folgen müßte, die wir in Kapitel 7 vorstellen werden. Die jeweilige Implementierung für die Markierung und Reduktion wird dort beschrieben.

Reduktion in den Teilausdrücken

Wenn nun beim Mustervergleich des zu reduzierenden (Sub-)Terms kein passender Fall (bzgl. Termstruktur oder Markierung) gefunden werden kann, dann muß in den entsprechenden Teiltermen mit der Reduzierung fortgefahren werden. Dies scheint bei einigen Termen zwar unsinnig, sollte aber möglich sein, weil wir auch Reduktionen betrachten wollen, die z. B. unterhalb von Abstraktionen reduzieren.

Keine unserer Reduktionen reduziert Variablen oder Konstruktoren, also werden keine markierten Terme erzeugt.

```
reduceCoreExpr rec level e@(EVar _ _)
    = []

reduceCoreExpr rec level e@(ECons _ _ _)
    = []
```

Die beiden Sub-Terme von `choice` werden beide nacheinander reduziert und die Ergebnismengen vereint. Normalerweise befindet sich nur in einem Teilterm der markierte Redex, so daß entweder beide oder genau eine Ergebnismenge leer ist.

```
reduceCoreExpr rec level e@(EChoice Unmarked e1 e2)
    =    [ EChoice Unmarked e1' e2
         | e1' <- reduceCoreExpr rec level e1
         ]
      ++ [ EChoice Unmarked e1 e2'
         | e2' <- reduceCoreExpr rec level e2
         ]
```

Innerhalb von Abstraktionen soll eigentlich nicht reduziert werden. Falls jedoch die Option erwünscht ist (siehe `markReduction`), dann muß auch in der Abstraktion fortgefahren werden.

```
reduceCoreExpr rec level (ELam  Unmarked v  e)
    = [ ELam Unmarked v e'
      | e' <- reduceCoreExpr rec level e
      ]
```

Beide Teilterme der Applikation werden ebenfalls nacheinander reduziert.

```
reduceCoreExpr rec level (EApp Unmarked e1 e2)
    =    [ EApp Unmarked e1' e2
         | e1' <- reduceCoreExpr rec level e1
         ]
      ++ [ EApp Unmarked e1 e2'
         | e2' <- reduceCoreExpr rec level e2
         ]
```

Der gesamte `let(rec)`-Ausdruck muß untersucht werden. Dazu werden sowohl die `let(rec)`-gebundenen Ausdrücke als auch der Term `e` geprüft und die Ergebnismengen vereint.

```
reduceCoreExpr rec level (ELet Unmarked r bs e)
    =    [ ELet Unmarked rec bs e'
         | e' <- reduceCoreExpr rec level e
         ]
      ++ [ ELet Unmarked rec bs' e
         | bs' <- concatMapOneItem reduceBinding bs
         ]
         where
           reduceBinding (v,expr)
             = [ (v,expr')
               | expr' <- reduceCoreExpr rec level expr
               ]
```

Auch innerhalb der Alternativen wird weiter reduziert, da einige Reduktionen auch in Oberflächenkontexten (siehe Definition 3.1.5, Seite 22) stattfinden können.

```
reduceCoreExpr rec level (ECase Unmarked tp e as)
    =    [ ECase Unmarked tp e' as
         | e' <- reduceCoreExpr rec level e
         ]
      ++ [ECase Unmarked tp e  as'
         | (t,vs,ae) <- as
         , ae'       <- reduceCoreExpr rec level ae
         , let as'    = updateAltExpr t ae' as
         ]
```

Die oben angegebenen Funktionsdefinitionen von `reduceCoreExpr` müssen alle möglichen Fälle abdecken, weshalb dieser Fall niemals eintreten sollte. Wenn aber doch, dann kann das folgende Ursachen haben:

- Für eine neu implementierte Reduktion wurden nicht beide Kalküle berücksichtigt.
- Der Reduktionsmechanismus erlaubt nur Terme mit den Markierungen `(Redex r)` und `VWHNF`. Terme mit anderen Marken können nicht reduziert werden.

```
reduceCoreExpr rec level e
    = error $ "Reduction.reduceCoreExpr: "
              ++ "wrong marked expression `" ++ show e ++ "`"
```

Kapitel 4

Kontextuelle Äquivalenz und Reduktionsdiagramme

Der im vorherigen Kapitel vorgestellte Kalkül $\lambda_{nd,rec}$ bzw. seine Sprachelemente verfügt schon gute Vorraussetzungen für eine nicht-strikte, funktionale Programmiersprache. Doch wie kann man die Eigenschaften und somit auch die Korrektheit solcher Kalküle genauer beschreiben?

Wenn wir die Eigenschaften von funktionalen Programmiersprachen oder Programmen, die in diesen Sprachen geschrieben sind, untersuchen wollen, dann wird meist die Gleichheit von Termen (Programmen) betrachtet. Man will also zwei verschiedene Terme dahingehend untersuchen, ob sie bei gleichen Eingaben dieselben Ergebnisse liefern. Es wird also versucht, den Termen eine Bedeutung zuzumessen, die dann verglichen werden kann. Dazu gibt es verschiedene Methoden:

- *Denotationale Semantik*
  ist der Versuch, eine mathematische Beschreibung (*domain*) und eine Abbildung von Ausdrücken der untersuchten Sprache in ihre Bedeutung zu finden. Dadurch kann eine Gleichheit von Ausdrücken beschrieben werden und somit die Gültigkeit von Programmtransformationen[1]. Mit dieser Methode konnte die Semantik einer Sprachen bisher erfolgreich beschrieben werden. Bei nichtdeterministischen Sprachen, die zusätzlich noch Rekursion möglich machen, hat diese Methode leider ihre Grenzen.
- *Operationale Semantik*
  beschreibt wie ein Term ausgewertet wird bzw. wie ein Programm ausgeführt wird. Anhand dieser Auswertungsreihenfolge können Terme verglichen werden und somit die Korrektheit bestimmter Auswertungsschritte (Programmtransformationen) bewiesen werden. In dieser Arbeit ist die Definition einer Auswertung in Normalordnung die operationale Semantik der Sprache.
- *Kontextuelle Semantik*
  ist eine operationale Semantik, die durch eine kontextuelle Präordnungsrelation erwei-

[1] Mit Programmtransformation ist die Ausführung eines Programms bzw. die Auswertung von Termen gemeint.

tert ist, die zu einer kontextuellen Äquivalenz führt. Wenn zwei Terme s und t kontextuell gleich sind bzw. sich gleich verhalten, dann kann man in allen Programmen (Programmkontexten) diese beiden Terme gegeneinander austauschen, ohne daß sich die Semantik des Programms verändert ([Pit97]). Dadurch ist es möglich, Programmtransformationen in einer Sprache auf ihre Korrektheit zu überprüfen.

4.1 Kontextuelle Äquivalenz

In diesem Kapitel werden wir die kontextuelle Präordnung und die kontextuelle Äquivalenz sowie ein Kontextlemma vorstellen, das es uns ermöglicht, von Reduktionskontexten auf allgemeine Kontexte zu schließen. Doch beginnen wir mit der Definition von Eigenschaften eines Ausdrucks (Terms).

Definition 4.1.1 Sei der Ausdruck t gegeben, dann schreiben wir:

$t\Downarrow_{WHNF}$ genau dann, wenn von t ausgehend eine NO-Reduktion zu einem Ausdruck in WHNF existiert.

$t\Downarrow_{CWHNF}$ genau dann, wenn von t ausgehend eine NO-Reduktion zu einem Ausdruck in CWHNF existiert.

$t\Downarrow_{VWHNF}$ genau dann, wenn von t ausgehend eine NO-Reduktion zu einem Ausdruck in VWHNF existiert.

$t\Downarrow$ genau dann, wenn $t\Downarrow_{WHNF}$ oder $t\Downarrow_{CWHNF}$ oder $t\Downarrow_{VWHNF}$.

$t\not\Downarrow$ genau dann, wenn von t ausgehend keine NO-Reduktion zu einem Ausdruck in WHNF existiert, d. h. alle von t ausgehenden NO-Reduktionen nicht terminieren.

$t\Downarrow\Downarrow$ genau dann, wenn von t ausgehend eine nichtterminierende NO-Reduktion existiert.

$t\not\Downarrow\not\Downarrow$ genau dann, wenn von t ausgehend keine nichtterminierende NO-Reduktion existiert, d. h. alle von t ausgehenden NO-Reduktionen terminieren bei einem Ausdruck in WHNF, CWHNF oder VWHNF.

An einigen Beispielen soll die oben eingeführte Notation verdeutlicht werden:

Beispiel 4.1.1

Ausdruck t	Eigenschaften der NO-Reduktion
$\lambda x.x$	$t\Downarrow_{WHNF}$, $t\Downarrow$, $t\not\Downarrow\not\Downarrow$
$(\lambda x.xx)(\lambda x.xx)$	$t\Downarrow\Downarrow$, $t\not\Downarrow$
`let` x `=` Nil `in` x	$t\Downarrow_{CWHNF}$, $t\Downarrow$, $t\not\Downarrow\not\Downarrow$
`let` x `=` y `in` x $y \in \mathcal{FV}(t)$	$t\Downarrow_{VWHNF}$, $t\Downarrow$, $t\not\Downarrow\not\Downarrow$
`choice` (`let` x `=` y `in` x) $(\lambda x.x)$	$t\Downarrow_{WHNF}$, $t\Downarrow_{VWHNF}$, $t\Downarrow$, $t\not\Downarrow\not\Downarrow$
`choice` (`let` x `=` y `in` x) $((\lambda x.xx)(\lambda x.xx))$	$t\Downarrow_{VWHNF}$, $t\Downarrow$, $t\Downarrow\Downarrow$

Wir wollen das Terminierungsverhalten zweier Terme vergleichen. Dazu wollen wir die Terme ordnen: d. h. ein Term s hat weniger Informationen (über sein Terminierungsverhalten) als ein zweiter Term t, genau dann, wenn für alle Kontexte, in die s und t eingesetzt werden, gilt, daß aus dem Terminierungsverhalten s das von t geschlossen werden kann.

Definition 4.1.2 (Kontextuelle Präordnung) Die *Kontextuelle Präordnung* ($\leq_C$) ist wie folgt definiert:

$$s \leq_C t \iff \begin{array}{l} \forall C[\,]\text{: wenn } C[s], C[t] \text{ geschlossen, dann} \\ (C[s]\Downarrow \;\Rightarrow\; C[t]\Downarrow) \;\wedge\; (C[s]\Downarrow\!\Downarrow \;\Rightarrow\; C[t]\Downarrow\!\Downarrow) \end{array}$$

Aus dieser Ordnungsrelation können wir sofort die kontextuelle Äquivalenz herleiten.

Definition 4.1.3 (Kontextuelle Äquivalenz) Die *Kontextuelle Äquivalenz* ($\sim_C$) ist:

$$s \sim_C t \iff s \leq_C t \;\wedge\; t \leq_C s$$

Wir müssen jetzt also nur beweisen, daß das Terminierungsverhalten zweier Terme in allen Kontexten gleich bleibt, um die beiden Terme als gleich anzusehen. Das ist jedoch einfacher gesagt als getan, da die Behauptung „in allen Kontexten“ nicht gerade trivial ist.

Viel besser wäre es, die Kontexte einzuschränken, d. h. Kontexte zu definieren, die einfacher zu untersuchen sind, und dann auf alle Kontexte zu schließen. Dabei hilft uns das Kontext-Lemma.

Lemma 4.1.1 (Kontext-Lemma) Seien s,t Terme und R ein Reduktionskontext:

$$\begin{array}{c} \forall R[\,]\text{: wenn } R[s], R[t] \text{ geschlossen, dann} \\ (R[s]\Downarrow \;\Rightarrow\; R[t]\Downarrow) \;\wedge\; (R[s]\Downarrow\!\Downarrow \;\Rightarrow\; R[t]\Downarrow\!\Downarrow) \end{array} \implies s \leq_C t$$

Beweis. Wir beweisen die allgemeinere Form der Behauptung:

> Wenn s_i, t_i die Bedingungen des Kontext-Lemmas erfüllen, dann auch für alle Multi-Kontexte $C[\cdot_1, \ldots, \cdot_m]$, so daß $C[s_1, \ldots, s_n] \Downarrow \;\Rightarrow\; C[t_1, \ldots, t_n] \Downarrow$ und $C[s_1, \ldots, s_n] \Downarrow\!\Downarrow \;\Rightarrow\; C[t_1, \ldots, t_n] \Downarrow\!\Downarrow$ gelten.

Wir nehmen an, daß diese Aussage falsch ist. Dann gibt es einen Multikontext C, so daß entweder $(C[s_1, \ldots, s_n]\Downarrow \wedge C[t_1, \ldots, t_n]\not\Downarrow)$ oder $(C[s1, \ldots, sn]\Downarrow\!\Downarrow \wedge C[t1, \ldots, tn]\not\Downarrow\!\not\Downarrow)$.

Wir wählen einen minimalen Kontext bzgl. der aus folgenden zwei Komponenten bestehenden Ordnung:

1. Die Anzahl der NO-Reduktionen, der kürzesten NO-Reduktion von $C[s_1, \ldots, s_n]$ zu WHNF.

2. Die Anzahl der Löcher im Kontext C.

1. $C[s_1, \dots, s_n]\Downarrow \wedge C[t_1, \dots, t_n]\not\Downarrow$

   Bei der Suche des NO-Redex gibt es zwei mögliche Fälle zu betrachten:

   (a) Einer der Kontexte $C[\cdot_1, s_2, \dots, s_n], \dots, C[s_1, \dots, s_{n-1}, \cdot_n]$ ist ein Reduktionskontext. Wir wählen o.b.d.A. den ersten Kontext $C[\cdot_1, \dots, \cdot_n]$. Dann ist $C[\cdot_1, t_2, \dots, t_n]$ ebenfalls ein Reduktionskontext. Wir definieren $C' := C[s_1, \cdot_2, \dots, \cdot_n]$. Wegen $C'[s_2, \dots, s_n] \equiv C[s_1, \dots, s_n]$, haben beide Kontexte dieselbe NO-Reduktion. Weil die Anzahl der Löcher in C' jedoch kleiner ist als die in C, erhalten wir gemäß der Voraussetzungen des Lemmas $C'[t_2, \dots, t_n]\Downarrow$ bzw. $C[s_1, t_2, \dots, t_n]\Downarrow$.

   Da aber $C[\cdot, t_2, \dots, t_n]$ ebenfalls ein Reduktionskontext ist, impliziert das Lemma $C[t_1, t_2, \dots, t_n]\Downarrow$. Die ist jedoch ein Widerspruch zur Annahme $C[t_1, t_2, \dots, t_n]\not\Downarrow$.

   (b) Keiner der Kontexte $C[\cdot_1, s_1, \dots, s_n], \dots, C[s_1, \dots, s_{n-1}, \cdot_n]$ ist ein Reduktionskontext. Dann können beide Kontexte $C[s_1, \dots, s_n]$ und $C[t_1, \dots, t_n]$ mittels derselben NO-Reduktion reduziert werden, da die Suche nach dem NO-Redex im umgebenden Kontext $C[\dots]$ und nicht in den Löchern stattfindet. Beim Reduzieren könnte ein Kontext $C'[\dots]$ entstehen, der mehr Löcher besitzt, wobei die Löcher durch Kopien von s_i, t_i gefüllt sind. Induktion über die Reduktionsschritte in Normalordnung zeigt, daß $C[t_1, \dots, t_n]\Downarrow$, also ein Widerspruch zu $C[t_1, \dots, t_n]\not\Downarrow$.

2. $C[s1, \dots, sn]\Downarrow\Downarrow \wedge C[t1, \dots, tn]\not\Downarrow\not\Downarrow$

   Analog zu 1.

Insgesamt konnte die Falschannahme widerlegt werden. □

Wir wollen Programm- oder Termtransformationen dahingehend überprüfen, daß sie in unseren Kalkülen gültig sind und das Ergebnis einer Auswertung nicht verändern. Eine solche Transformation wäre beispielsweise das Löschen einer nicht mehr referenzierten `let`-Bindung: `let` x `= Nil in True` $\rightarrow$ `True`. Diese Transformation werden wir in Kapitel 7.2.2 genauer betrachten. Doch zunächst die Definition für eine Programmtransformation.

Definition 4.1.4 (Programmtransformation) Eine *Programmtransformation* ist eine Relation T zwischen Termen (Programmen). Eine Programmtransformation T gilt als *korrekt*, wenn für alle Terme t und t' mit $(t\,T\,t')$ die kontextuelle Äquivalenz $t \sim_C t'$ gilt.

Prinzipiell stellt diese Definition die Einhaltung der referenziellen Transparenz sicher (siehe Kapitel 2.3.2, Seite 12).

Wie schon oben angedeutet werden wir in dieser Arbeit die Programmtransformationen wie *normale* Reduktionsregeln implementieren, obwohl wir die Kalküle nicht durch diese neuen Reduktionsregeln erweitert sehen wollen. Vielmehr erleichtert uns diese Gleichstellung den Umgang mit ihnen.

Definition 4.1.5 (Interne Reduktion) Eine *interne Reduktion* (a) ist eine *nicht*-NO-Reduktion, die sich in einem Reduktionskontext befindet. Wir bezeichnen dazu (i,a) als interne und (no,a) als NO-Reduktion.

Beispiel 4.1.2 Der Term ((`let` x `= Succ in` x) `Zero`) ist ein (*lapp*)-Redex und ein (*i,cp*)-Redex.

$$\begin{array}{lcl}
(\underline{(\texttt{let } x \texttt{ = Succ in } x) \texttt{ Zero}}) & \xrightarrow{no,lapp} & (\texttt{let } x \texttt{ = Succ in } (x \texttt{ Zero})) \\
((\underline{\texttt{let } x \texttt{ = Succ in } x}) \texttt{ Zero}) & \xrightarrow{i,cp} & ((\texttt{let } x \texttt{ = Succ in Succ}) \texttt{ Zero})
\end{array}$$

Die Frage, ob alle in den Definitionen 3.1.9 und 3.1.10 aufgeführten Reduktionsregeln auch intern in Reduktionskontexten vorkommen, beantwortet folgendes Lemma.

Lemma 4.1.2 Eine interne Reduktion ist vom Typ (*llet*) oder (*cp*). D. h. jede *(a)-Reduktion* mit $a \in \{lbeta, abs, lapp, lcase, case, ndl, ndr\}$ ist eine NO-Reduktion.

Beweis. Betrachtung aller Fälle. Man beachte, daß die Definition der (*abs*)-Reduktion (Abstraktion von links nach rechts) interne (*abs*)-Reduktionen verhindert. □

Wir wollen nun die Korrektheit der Programmtransformationen unserer Kalküle zeigen, wobei wir die Reduktionsregeln (*llet*), (*cp*) und die nichtdeterministischen Regeln (*ndl*) und (*ndr*) später behandeln wollen.

Behauptung 4.1.1 Wenn $s \xrightarrow{a} t$ mit $a \in \{lbeta, abs, lapp, lcase, case\}$, dann $s \sim_C t$. Also, jede Programmtransformation, die durch die Reduktionsregeln (*lbeta*), (*abs*), (*lapp*), (*lcase*) oder (*case*) beschrieben wird, ist korrekt im Sinne von Definition 4.1.4.

Beweis. Wir betrachten die Reduktion $s' \xrightarrow{a} t'$ innerhalb eines Lochs von s. Die zwei folgenden Fälle sind zu prüfen:

1. $R[s']\Downarrow \Rightarrow R[t']\Downarrow$
   Sei R ein Reduktionskontext. Dann sind die internen Reduktionen $R[s'] \xrightarrow{i,a} R[t']$ gemäß Lemma 4.1.2 nicht möglich. Also ist $R[s'] \xrightarrow{no,a} R[t']$ eine eindeutig definierte Reduktion. Wenn es also eine terminierende Reduktion für $R[s']$ gibt, dann auch für $R[t']$. Wenn es andererseits eine terminierende Reduktion für $R[t']$ gibt, dann auch für $R[s']$.

2. $R[s']\Downarrow\Downarrow \Rightarrow R[t']\Downarrow\Downarrow$
   Alle NO-Reduktionen von $R[s']$ terminieren dann und nur dann, wenn alle NO-Reduktionen von $R[t']$ terminieren.

Dadurch ergibt sich einerseits $s' \leq_C t'$, andererseits $t' \leq_C s'$ und daraus $s' \sim_C t'$. Da $\sim_C$ eine Kongruenz-Relation ist, ergibt sich insgesamt für einen Kontext C die Behauptung $C[s'] \sim_C C[t']$, also auch $s \sim_C t$. □

4.2 Reduktionsdiagramme

Reduktionsdiagramme beschreiben Zusammenhänge zwischen verschiedenen Reduktionen eines Terms. Wir möchten mit einem Reduktionsdiagramm beschreiben, wie eine bestimmte Reduktionsfolge durch eine andere Reduktionsfolge ersetzt werden kann. Wir benutzen die in [Kut00] beschriebenen Begriffe.

Definition 4.2.1 (Reduktionsdiagramm) Ein *Reduktionsdiagramm* ist ein gerichteter Graph, dessen Knoten Ausdrücke sind und dessen Kanten Reduktionen entsprechen und markiert sein können. Die Kanten können folgende Form haben:

Ein Pfeil stellt eine gegebene Reduktion dar.
Ein gestrichelter Pfeil stellt eine existenzquantifizierte Reduktion dar.

Die Markierungen der Kanten sind die Reduktionstypen, z. B. (*i,llet*) für eine interne (*llet*)-Reduktion.

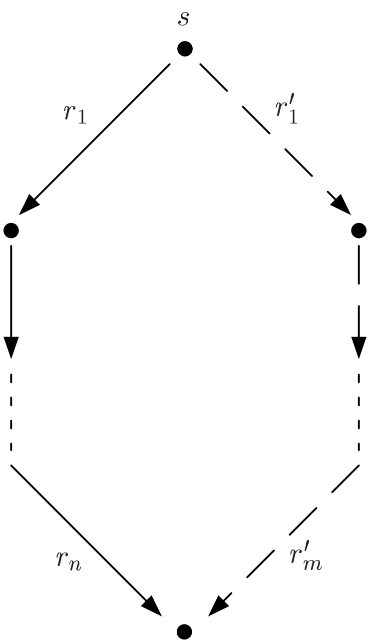

Abbildung 4.1: Vertauschungsdiagramm

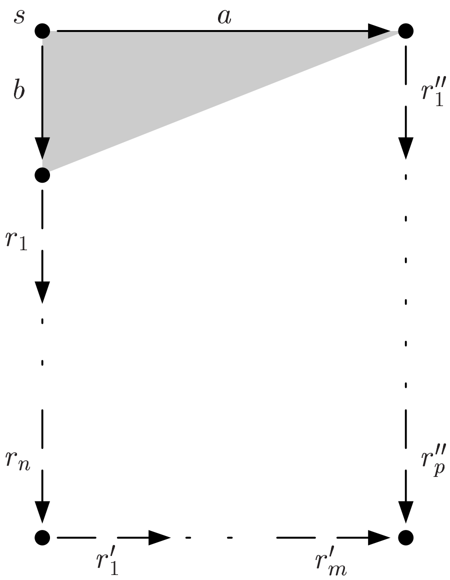

Abbildung 4.2: Gabeldiagramm

4.2.1 Vertauschungsdiagramme

Mittels Vertauschungsdiagrammen können wir eine bestimmte Reduktionsfolge durch eine andere ersetzen, wenn das Ergebnis (das Redukt) beider Reduktionen das gleiche ist. Das Vertauschungsdiagramm bildet also eine Meta-Reduktionsregel, also eine Regel für Reduktionsfolgen.

Definition 4.2.2 Ein Vertauschungsdiagramm ist eine Meta-Reduktionsregel der Form

$$r_1 \circ \cdots \circ r_n \quad \rightsquigarrow \quad r'_1 \circ \cdots \circ r'_m \qquad \text{mit } n \geq 2,\, m \geq 0$$

Der entsprechende Graph ist in Abbildung 4.1 aus Seite 60 zu sehen.

Beispiel 4.2.1 Wir greifen das Beispiel von Seite 59 auf und reduzieren $s \xrightarrow{i,cp} s' \xrightarrow{no,cp} s''$ und $s \xrightarrow{no,lapp} t' \xrightarrow{no,cp} t''$.

$$
\begin{array}{ll}
 & ((\underline{\texttt{let}\ x\ \texttt{= Succ in}\ x})\ \texttt{Zero}) \\
\xrightarrow{i,cp} & (\underline{(\texttt{let}\ x\ \texttt{= Succ in Succ})\ \texttt{Zero}}) \\
\xrightarrow{no,lapp} & (\texttt{let}\ x\ \texttt{= Succ in (Succ Zero)})
\end{array}
$$

$$
\begin{array}{ll}
 & (\underline{(\texttt{let}\ x\ \texttt{= Succ in}\ x)\ \texttt{Zero}}) \\
\xrightarrow{no,lapp} & (\underline{\texttt{let}\ x\ \texttt{= Succ in}\ (x\ \texttt{Zero}})) \\
\xrightarrow{no,cp} & (\texttt{let}\ x\ \texttt{= Succ in (Succ Zero)})
\end{array}
$$

Das Ergebnis beider Reduktionen ist derselbe Term ($s'' \equiv t''$), so daß wir folgendes Vertauschungsdiagramm angeben können:

$$(i, cp) \circ (no, lapp) \quad \rightsquigarrow \quad (no, lapp) \circ (no, cp)$$

Das Beispiel zeigt, daß die beiden Reduktionen vertauscht werden können, ohne das Ergebnis der Reduktion zu verändern. Eine weitere Eigenschaft ist, daß die interne (cp)-Reduktion nicht nur den Platz mit der ($lapp$)-Reduktion tauscht, sondern auch noch zu einer NO-Reduktion wird. Wir können — und wollen — also mittels Vertauschungsdiagrammen eine Reduktionsfolge derart umordnen, so daß zuerst alle NO-Reduktionen in der Folge auftreten und die internen Reduktionen nach rechts „geschoben“ werden.

Vollständiger Satz von Vertauschungsdiagrammen

Da meist kein universelles Vertauschungsdiagramm angegeben werden kann, das auf alle Terme einer Sprache anwendbar ist, müssen wir in vielen Fällen mehrere Diagramme angeben, die wir zu einem Satz zusammenfassen.

Definition 4.2.3 Ein *vollständiger Satz von Vertauschungsdiagrammen* für die Reduktionsart (a) (z. B. ($llet$)) ist eine Menge von Reduktionsdiagrammen $\mathcal{V}$ mit folgenden Eigenschaften:

Für alle Paare von Ausdrücken (t, t') mit $t \xrightarrow{a} t'$ existiert ein Vertauschungsdiagramm $V \in \mathcal{V}$ sowie ein Ausdruck t'', so daß V ein Reduktionsdiagramm R beschreibt, bei dem eine Ersetzung aller Platzhalter derart möglich ist, daß die gegebene Reduktion des Reduktionsdiagramms R die Form $t \xrightarrow{a} t' \xrightarrow{no+} t''$.

Das bedeutet, daß für alle Terme einer Sprache bzw. für alle Reduktionsfolgen der Terme ein Vertauschungsdiagramm in dem Satz enthalten sein muß, damit er vollständig ist. Unser System JONAH generiert dazu eine große Anzahl an Termen und prüft, ob eines der Vertauschungsdiagramme des Satzes auf den Term anwendbar ist.

4.2.2 Gabeldiagramme

Zusätzlich zu den Vertauschungsdiagrammen benötigen wir noch eine andere Form von Reduktionsdiagrammen, nämlich Gabeldiagramme.

Definition 4.2.4 Ein Gabeldiagramm ist eine Meta-Reduktionsregel der Form

$$\xleftarrow{r_n} \circ \cdots \circ \xleftarrow{r_1} \circ \xleftarrow{b} \circ \xrightarrow{a} \quad \rightsquigarrow \quad \xrightarrow{r'_1} \circ \cdots \circ \xrightarrow{r'_m} \circ \xleftarrow{r''_p} \circ \cdots \circ \xleftarrow{r''_1}$$

$$\text{mit } m,\, n,\, p \geq 0$$

Der entsprechende Graph ist in Abbildung 4.2 zu sehen.

Bei Gabeldiagrammen ist nicht die Folge von Reduktionen interessant, sondern die Verzweigung, also die beiden Ausgangsreduktionen. Unser Beispiel-Term ((`let` x `= Succ in` x) `Zero`) hätte somit das folgende Gabeldiagramm:

$$\xleftarrow{no,lapp} \circ \xrightarrow{i,cp} \quad \rightsquigarrow \quad \xrightarrow{no,cp} \circ \xleftarrow{no,lapp}$$

Vollständiger Satz von Gabeldiagrammen

Definition 4.2.5 Ein *vollständiger Satz von Gabeldiagrammen* für die Reduktionsart (a) (z. B. $(llet)$) ist eine Menge von Reduktionsdiagrammen $\mathcal{G}$ mit folgenden Eigenschaften:

Für alle Tripel von Ausdrücken (t, t', t'') mit

- t befindet sich nicht in WHNF
- $t \xrightarrow{a} t'$ und $t \xrightarrow{no} t''$

existiert ein Gabeldiagramm $G \in \mathcal{G}$ sowie eine Ersetzung aller Platzhalter in dem durch G beschriebenen Reduktionsdiagramm R, so daß die Gabel des Reduktionsdiagramms R nach Ersetzung aller Platzhalter $t'' \xleftarrow{no} t \xrightarrow{a} t'$ entspricht.

4.2.3 Allgemeine Vertauschungs- und Gabeldiagramme

Damit die Sätze von Vertauschungs- und Gabeldiagrammen nicht zu überladen sind, erlauben wir bestimmte Schreibweisen, die verallgemeinerte Formen dieser Diagramme ermöglichen. Wir wollen folgende besonderen Darstellungsformen innerhalb von Vertauschungs- und Gabeldiagrammen erlauben:

V	G	Vertauschungsdiagramme (Gabeldiagramme analog)
r^*	$\xrightarrow{r^*}$	Die Reduktion kann nicht oder mehrmals hintereinander auftreten, z. B. $r \circ r \circ r \circ r$.
r^+	$\xrightarrow{r^+}$	Die Reduktion kann mindestens einmal oder mehrmals hintereinander auftreten.
$r^{0 \vee 1}$	$\xrightarrow{r^{0 \vee 1}}$	Die Reduktion kann genau einmal oder gar nicht auftreten.
$k, \{r_1, \ldots, r_n\}*$	$\xrightarrow{k, \{r_1, \ldots, r_n\}*}$	Für $k \in \{i, no\}$ ist eine Folge von Reduktionen mit Länge ≥ 0 und den Reduktionen aus $\{r_1, \ldots, r_n\}$ möglich. Die Reduktionen müssen alle denselben Typ (i oder no) haben. Z. B. $(no, r_2) \circ (no, r_1) \circ (no, r_4)$ oder $(i, r_5) \circ (i, r_3) \circ (i, r_1)$.
$k, \{r_1, \ldots, r_n\}+$	$\xrightarrow{k, \{r_1, \ldots, r_n\}+}$	Wie oben, jedoch mit einer Mindestlänge von 1.

4.2.4 Beziehung zwischen Vertauschungs- und Gabeldiagrammen

Unser Beispiel-Term ((`let` x `= Succ in` x) `Zero`) zeigt, daß es zumindest für einfache Diagramme möglich ist, aus einem Vertauschungsdiagramm ein Gabeldiagramm herzuleiten. In diesem Fall liegt das an der Symmetrie der Reduktionsfolgen. In Abbildung 4.3 beschreiben die äußeren Pfeile das Vertauschungsdiagramm, die inneren das Gabeldiagramm.

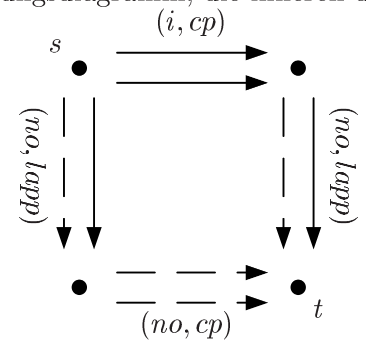

Abbildung 4.3: Beziehung zwischen Reduktionsdiagrammen

Es gibt jedoch auch Gabeldiagramme, aus denen sich keine Vertauschungsdiagramme herleiten lassen:

$$\xleftarrow{r_1} \circ \xrightarrow{r_2} \quad \rightsquigarrow \quad \xrightarrow{r_3}$$

würde das Vertauschungsdiagramm

$$r_2 \quad \rightsquigarrow \quad r_1 \;\circ\; r_3$$

ergeben, welches nach der Definition von Vertauschungsdiagrammen nicht zulässig ist.

Wir werden jedoch in unserem System auch solche „falschen“ Vertauschungsdiagramme validieren können. Für die Beweise können sie allerdings nicht benutzt werden.

Den eigentlichen Unterschied zwischen Vertauschungs- und Gabeldiagramm bilden die gegebenen und die existenzquantifizierten Reduktionstypen. Dies ist bei der Verwendung in den Beweisen natürlich zu beachten. Für unsere in Kapitel 6 vorgestellte Validierung kann dieser Unterschied jedoch wegfallen, da wir das Validierungsverfahren genau auf beide Typen von Reduktionsdiagrammen abgestimmt haben. Daraus ergibt sich jedoch, daß bei einer Änderung der Struktur von Reduktionsdiagrammen in Bezug auf die Quantifizierung, das Verfahren nicht mehr gültig ist.

Kapitel 5

Termgenerierung

Zum Testen der Reduktionsdiagramme sollen aus der unendlichen Menge aller Terme Teilmengen generiert werden. Diese sollten wohlgeformt sein, jedoch müssen sie nicht unbedingt wohlgetypt sein. Wir verzichten bewußt auf die Generierung von wohlgetypten Ausdrücken, da die Typ-Überprüfung zusätzlich Zeit kosten würde. Es ist jedoch möglich, das Programm so zu erweitern, daß der Termgenerierung eine Typüberprüfung nachgeschaltet wird, die nur wohlgetypte Ausdrücke zuläßt. In unserem System werden alle Ausdrücke so lange reduziert bis nicht mehr reduziert werden kann oder ein Fehler auftritt.

5.1 Aufzählen der Ausdrücke

Definition 5.1.1 Die Menge $\mathcal{T}(V,C) \subset \Lambda_c$ ist die Menge aller Terme, die Variablen aus V und Konstruktoren aus C enthalten, wobei V und C endliche Mengen sind.[1]

Wegen $|\mathcal{T}(V,C)| = \infty$, müssen wir die Aufzählung begrenzen. Dazu bilden wir disjunkte Teilmengen $\mathcal{T}_x(V,C) \subset \mathcal{T}(V,C)$, wobei gilt:

$$\bigcup_{x \to \infty} \mathcal{T}_x(V,C) = \mathcal{T}(V,C)$$

Die Teilmengen $\mathcal{T}_x(V,C)$ enthalten nur Terme, die alle dieselbe Anzahl von Sprachelementen besitzen:

Definition 5.1.2 Das 5-Tupel $f(T) = (lam, app, let, case, choice)$ bestimme die Anzahl der Abstraktionen, Applikationen, `let(rec)`-, `case`- und `choice`-Ausdrücke im Term T, wobei gilt:

$$\begin{aligned} f(M) + f(N) \quad = \quad & (lam_M + lam_N, app_M + app_N, let_M + let_N, \\ & case_M + case_N, choice_M + choice_N) \end{aligned}$$

[1] $\mathcal{T}(V,C)$ soll nur für die Validierung interessante Terme enthalten. Deshalb werden einige Terme a priori ausgeschlossen (siehe Kapitel 5.2).

Definition 5.1.3 Wir bezeichnen die Menge aller Terme, die Variablen aus V und Konstruktoren aus C sowie Abstraktionen, Applikationen, `let(rec)`-, `case`- und `choice`-Ausdrücke mit den Häufigkeiten $(lam, app, let, case, choice)$ enthalten, als $\mathcal{T}_{(lam,app,let,case,choice)}(V,C)$ und definieren diese rekursiv:

$$
\begin{array}{rcl}
\mathcal{T}_{(0,0,0,0,0)}(V,C) & = & V \cup C \\
\mathcal{T}_{class}(V,C) & = & \bigcup \mathcal{T}^{x}_{class}(V,C) \quad \text{mit } x \in \{lam, app, let(rec), case, choice\}
\end{array}
$$

$$
\begin{array}{rcll}
\mathcal{T}^{lam}_{(lam,app,let,case,choice)}(V,C) & = & \{ & \lambda x\,.\,E \\
 & & | & E \in \mathcal{T}_{(lam-1,app,let,case,choice)}(V \cup \{x\}, C) \\
 & & \} & \\
\mathcal{T}^{app}_{(lam,app,let,case,choice)}(V,C) & = & \{ & E_1\ E_2 \\
 & & | & E_1, E_2 \in \mathcal{T}(V,C) \\
 & & , & f(E_1) + f(E_2) = (lam, app-1, let, case, choice) \\
 & & \} & \\
\mathcal{T}^{let}_{(lam,app,let,case,choice)}(V,C) & = & \{ & \texttt{let}\ x_1\texttt{=}E_1\ \texttt{in}\ E \\
 & & | & E_1 \in \mathcal{T}(V,C) \\
 & & , & E\ \in \mathcal{T}(V \cup \{x\}, C) \\
 & & , & f(E) + f(E_1) = (lam, app, let-1, case, choice) \\
 & & \} & \\
\mathcal{T}^{letrec}_{(lam,app,let,case,choice)}(V,C) & = & \{ & \texttt{letrec}\ x_1\texttt{=}E_1\ \ldots\ x_n\texttt{=}E_n\ \texttt{in}\ E \\
 & & | & E_i \in \mathcal{T}(V \cup \{x_1,...,x_n\}, C) \\
 & & , & E\ \in \mathcal{T}(V \cup \{x_1,...,x_n\}, C) \\
 & & , & f(E) + \sum_{i=1}^{n} f(E_i) = (lam, app, let-1, case, choice) \\
 & & \} & \\
\mathcal{T}^{case}_{(lam,app,let,case,choice)}(V,C) & = & \{ & \texttt{case}\ E\ \texttt{of}\ \langle c_1\ \vec{x}_1\texttt{->}E_1, \ldots, c_n\ \vec{x}_n\texttt{->}E_n\rangle \\
 & & | & E\ \in \mathcal{T}(V,C) \\
 & & , & E_i \in \mathcal{T}(V \cup \{\vec{x}_i\}, C) \\
 & & , & f(E) + \sum_{i=1}^{n} f(E_i) = (lam, app, let, case-1, choice) \\
 & & \} & \\
\mathcal{T}^{choice}_{(lam,app,let,case,choice)}(V,C) & = & \{ & \texttt{choice}\ E_1\ E_2 \\
 & & | & E_1, E_2 \in \mathcal{T}(V,C) \\
 & & , & f(E_1) + f(E_2) = (lam, app, let, case, choice-1) \\
 & & \} &
\end{array}
$$

Die Funktion `generateAllCoreExprs` zählt alle Ausdrücke der Kernsprache bis zu einer maximalen Anzahl (`termClass`) von Abstraktionen, Applikationen, `let(rec)`- und `case`-Ausdrücken auf. Weitere Argumente von `generateAllCoreExprs` sind ein Schalter zur Bestimmung des verwendeten Kalküls (`rec` für *recursive*), die maximale Anzahl von Bindungen in `letrec`-Ausdrücken (`maxb`[2]), die zu verwendenden freien Variablen (`vs`) und abstrakten Datentypen (`ds`). `vname` gibt den nächsten, noch unbelegten, Namen neuer freier Variablen

[2]Im nicht-rekursiven Kalkül wird `maxb` Parameter ignoriert

an. Der Parameter `classonly` bestimmt, ob alle Terme einer bestimmten Klasse erzeugt werden, oder auch alle Unterklassen.

```
generateCoreExprs rec maxb classonly termClass varsAndCons expr
    | classonly = [ correctCoreExpr $ replaceVar "" e' expr
                  | e' <- genCoreExprs termClass varsAndCons settings
                  ]
    | otherwise = [ correctCoreExpr $ replaceVar "" e' expr
                  | termclass <- allSubClasses termClass
                  , e' <- genCoreExprs termclass varsAndCons settings
                  ]
                  where
                    settings = if expr == holeExpr -- nur das Loch
                                  then (True,0,True ,"",rec,"", maxb)
                                  else (True,0,False,"",rec,"", maxb)
```

`generateAllCoreExprs` ruft ihrerseits die Funktion `genCoreExprs` auf, die Ausdrücke mit einer *festen* Anzahl von Sprachelementen erzeugt. Die erzeugten Ausdrucksmengen werden dann zu einer einzigen Menge vereinigt. `genCoreExprs` generiert dazu je eine Menge von Ausdrücken, deren Wurzel eines der Sprachelemente (λ-*Ausdruck*, *Applikation*, `let`(`rec`)-, `case`-*Ausdruck* oder `choice`-*Ausdruck*) ist. Die jeweiligen Unterausdrücke werden dann aus allen möglichen Kombinationen rekursiv ermittelt. Mittels des Tupels `settings` wird direkt auf die Termgenerierung Einfluß genommen, um bestimmte Terme auszufiltern (siehe Kapitel 5.2). Die Funktionen `correctCoreExpr` und `replaceVar` werden weiter unten behandelt (siehe Kapitel 5.3, Seite 74).

5.1.1 λ-Ausdrücke

Für einen λ-Ausdruck muß eine neue Variable erzeugt werden, die beim rekursiven Aufruf übergeben wird. Der Parameter `toplevel` gewährleistet, daß λ-Ausdrücke nur in Teilausdrücken vorkommen.

```
genCoreExprs termClass@  (cLam, cApp, cLet, cCase, cChoice)
             varsAndCons@(vs, vname, ds)
             switches@   (toplevel, carity, csat, ctype, rec, recvar, maxb)
    = esLam ++ esApp ++ esLet ++ esCase ++ esChoice
        where
          lamLet           = if cLet > 2 then cLet - 2 else 0
          tcLam            = termClass - (1,0,lamLet,0,0)
          newVarsAndCons   = (vname:vs, nextVar vname, ds)
          esLam | cLam <= 0 = []
                | toplevel  = []
                | otherwise = [ ELam Unmarked vname e
                              | e  <- genCoreExprs  tcLam
                                                    newVarsAndCons
                                                    settingsE
                              ]
```

5.1.2 Applikationen

Bei Applikationen müssen die beiden Teilausdrücke `e1` und `e2` zusammengenommen in der Termklasse `tcApp` liegen. Der Term `e1` muß in allen Subklassen von `tcApp` liegen, so daß `e2` in der komplementären Termklasse (`tcApp - tc`) von `e1` sein muß.

```
tcApp              = termClass - (0,1,0,0,0)
esApp | cApp <= 0 = []
      | otherwise = [ EApp Unmarked e1 e2
                    | tc <- allSubClasses tcApp
                    , e1 <- genCoreExprs  tc
                                          varsAndCons
                                          settingsE1
                    , e2 <- genCoreExprs  (tcApp - tc)
                                          varsAndCons
                                          settingsE2
                    ]
```

Mittels des Parameters `settings` werden dem rekursiven Aufruf von `genCoreExprs` Optimierungsoptionen übergeben, so daß z. B. für `e1` kein nullstelliger Konstruktor generiert wird.

5.1.3 `let(rec)`-Ausdrücke

Bei `let`-Ausdrücken werden ebenfalls zwei Ausdrucksmengen (die einen `let(rec)`-Ausdruck weniger besitzen) erzeugt, deren Elemente dann miteinander kombiniert werden. .

```
tcLet              = termClass - (0,0,1,0,0)
esLet | cLet <= 0 = []
      | not rec   = [ ELet Unmarked rec [(vname,e1)] e2
                    | tc <- allSubClasses tcLet
                    , e1 <- genCoreExprs  tc
                                          varsAndCons
                                          settingsE1
                    , e2 <- genCoreExprs  (tcLet - tc)
                                          newVarsAndCons
                                          settingsE2
                    ]
```

Beim rekursiven Kalkül kann ein `letrec`-Ausdruck keine oder mehrere Bindungen besitzen. Deshalb muß durch den Parameter `maxb` die obere Grenze angegeben werden. Die Funktion `genNewVars` errechnet alle möglichen bzw. nötigen Variablenkombinationen, die mittels `genBindings` zu einer Bindungsliste, deren Terme zusammengenommen in der Klasse `tc` liegen, kombiniert werden. Der Ausdruck `e` kann dann nur noch in der Termklasse (`tcLet - tc`) sein.

```
              | otherwise = [ ELet Unmarked rec bs e
                            | tc <- allSubClasses tcLet
                            , nv <- genNewVars
                            , let vc = (nv ++ vs, nextVar (head nv), ds)
                            , bs <- genBindings  tc nv vc
                            , e  <- genCoreExprs (tcLet - tc)
                                                 vc
                                                 settingsE
                            ]
```

Da die Reihenfolge der Bindungen willkürlich sein kann, muß bei der Generierung darauf geachtet werden, daß nicht redundante Terme erzeugt werden. Dazu werden die Bindungen in einer Reihenfolge erzeugt, die die lexikographische Ordnung der Termklasse bildet. Beispielsweise sind die folgenden Terme — nach Umbenennen der Variablen — gleich:

```
let x=Cons Zero, y=Nil       in x y
let x=Nil,       y=Cons Zero in y x
```

Da der `in`-Term variiert, und somit auch alle Kopierziele der Variablen, reicht es, die `letrec`-Bindungen bzgl. der Termklasse der gebundenen Terme und des Variablennamens zu ordnen (`orderedClassCombinations`).

```
     genNewVars
       = gnv maxb [] vname
           where
             gnv 0 vs _ = []
             gnv n vs v = let vs' = (v:vs)
                                 in vs':(gnv (n-1) vs' (nextVar v))

     genBindings tc nv vc
       = concat [combine x | x <- bindings]
           where
             combine [] = [[]]
             combine ((v,vtc):xs)
               = [ (v,e):rest
                 | e    <- genCoreExprs vtc
                                        vc
                                        settingsB
                 , rest <- combine xs
                 ]
             bindings = [ zip nv combis
                        | combis <- orderedClassCombinations (length nv) tc
                        ]
```

5.1.4 `case`-Ausdrücke

`case`-Ausdrücke bestehen aus einem zu prüfenden Ausdruck und einer Menge von Alternativen, die sich aus der Definition der abstrakten Datentypen ergibt. Wir erzeugen also wiederum eine Menge von Ausdrücken (`e`) und eine Menge von Ausdrucksmengen (`as`). Die Anzahl der Elemente von `as` ist gleich der Anzahl der Konstruktoren des jeweiligen Typs. `genCoreExprs` versucht dabei nur Ausdrücke zu generieren, die vom gleichen Typ wie die

Alternativen sind. Dieser wird durch die Variable `ctype` angegeben. Natürlich muß auch hier sichergestellt sein, daß die resultierenden Ausdrücke alle die gleiche Anzahl an Sprachelementen besitzen. Dies übernimmt die Funktion `genCoreAlts`, die alle möglichen Kombinationen der komplementären Termklasse (`tcCase - tc`) erzeugt.

```
tcCase              = termClass - (0,0,0,1,0)
esCase | cCase <= 0 = []
       | otherwise  = [ ECase Unmarked ctype' e as
                      | (ctype', cs) <- ds
                      , tc <- allSubClasses tcCase
                      , e  <- genCoreExprs  tc
                                            varsAndCons
                                            settingsE
                      , as <- genCaseAlts ctype cs (tcCase - tc)
                      ]

genCaseAlts t cs termclass
  = concat [ combine x | x <- alts]
      where
        combine [] = [[]]
        combine ((t',vs,tc,vc,ps):xs)
          = [ (t',vs,e):rest
            | e    <- genCoreExprs tc vc ps
            , rest <- combine xs
            ]
        alts = [ [ ( tag
                   , vars
                   , termclass'
                   , (vs ++ vars, vname, ds)
                   , settingsA
                   )
                 | i <- [0..(length cs) - 1]
                 , let (tag, ar)  = cs !! i
                 , let vars       = take ar (nextVars "@")
                 , let termclass' = combis !! i
                 ]
               | combis <- classCombinations (length cs) termclass
               ]
```

5.1.5 `choice`-Ausdrücke

`choice`-Ausdrücke werden genauso generiert wie Applikationen, jedoch mit dem Unterschied, daß die Optimierungs-Schalter (`settings`) unverändert übernommen werden können, da die Stellung beider Teilausdrücke innerhalb des umgebenden Kontextes dieselbe ist wie die des `choice`.

```
tcChoice                = termClass - (0,0,0,0,1)
esChoice | cChoice <= 0 = []
         | otherwise    = [ EChoice Unmarked e1 e2
                          | tc <- allSubClasses tcChoice
                          , e1 <- genCoreExprs tc
                                               varsAndCons
                                               switches
                          , e2 <- genCoreExprs (tcChoice - tc)
                                               varsAndCons
                                               switches
                          ]
```

5.1.6 Variablen und Konstruktoren

Die Werte unseres Kalküls sind Variablen, Konstruktoranwendungen und λ-Ausdrücke. Diese sind also Ausdrücke, die weder `let(rec)`-, `case`- noch `choice`-Ausdrücke enthalten. Die Menge der Variablen besteht sowohl aus den Variablen, die man der Funktion `genCoreExprs` erstmalig übergeben kann, als auch aus den während der Termerzeugung eingeführten Variablen.

Wird ein gesättigter Konstruktor auf einen Ausdruck angewandt, z.B. `(Cons a Nil) b`, dann kann dieser Term nicht reduziert werden. Aus diesem Grund wird anhand des Arguments `carity` der Sättigungsgrad des Konstruktors vorgegeben. Sollen nämlich die erzeugten Ausdrücke auf einen weiteren Ausdruck angewandt werden, so werden nur ungesättigte Konstruktoranwendungen in die Menge aufgenommen.

```
genCoreExprs (0, 0, 0, 0, 0)
             (vs, vname, ds)
             (toplevel, carity, csat, ctype, rec, recvar, maxb)
    =    [ EVar Unmarked v
         | v <- vs
         , v /= recvar
         ]
      ++ [ ECons Unmarked t a
         | (ctype', cs) <- ds
         , (ctype == "") || (ctype == ctype')
         , (t, a) <- cs
         , if csat then carity == a else carity <= a
         ]
```

Die Parameter `csat` und `ctype` bestimmen, ob die zu erzeugenden Konstruktoren gesättigt bzw. welchen Typs sie sein sollen. Die Argumente `recvar` und `maxb` werden nur bei der Generierung von `letrec`-Ausdrücken (Parameter `rec`) benötigt. `recvar` ist der Variablenname, an den der Ausdruck gebunden wird, so daß pathologische Terme wie `let x=x in x` nicht erzeugt werden[3].

Beispiel:

[3] Terme wie `(letrec x=Succ x in x)` stellen zwar auch einen Ausdruck mit unendlicher Reduktion dar, jedoch sind solche rekursiven Datentypen keine so große *Gefahr*. Die Variable `x` bindet in diesem Fall die Zahl *unendlich*, die innerhalb eines Zahlenvergleichs ohne weiteres benutzt werden kann: `(letrec x=Succ x in less_or_equal Zero x)`

```
Main> genCoreExprs (0,0,0,0,0) (["a","b"],"c",ds) (True,0, True,"",False,"",0)
[a,b,Nil,True,False,Zero]
```

Die Parameter `toplevel`, `carity`, `csat` und `ctype` bestimmen welche Ausdrücke erzeugt bzw. nicht erzeugt werden. Eine genaue Beschreibung folgt weiter unten in Kapitel 5.2.

5.1.7 Hilfsfunktionen

Um alle Unterklassen zu generieren, wird das Kreuzprodukt über die maximale Anzahl der fünf Sprachelemente gebildet.

```
allSubClasses (a,b,c,d,e)
    = cross5 [0..a] [0..b] [0..c] [0..d] [0..e]
```

Die Funktion `cross5` erzeugt aus den fünf Argument-Listen eine Liste aller möglichen 5-Tupel, die in unserem Fall Zahlen als Komponenten enthalten.

```
cross5 as bs cs ds es
    = [(a, b, c, d, e) | a <- as, b <- bs, c <- cs, d <- ds, e <- es]
```

Bei den Alternativen eines `case`-Ausdrucks muß die Summe der Termklassen aller Teilterme gleich sein. `classCombinations` erzeugt alle `n`-elementigen Listen von 5-Tupeln, deren Summe `tc` ist. Die Listen sind eindeutig durch die Reihenfolge ihrer Elemente (5-Tupel) identifizierbar. D. h. die Listen `[(1,0,0,0,0),(0,1,0,0,0)]` und `[(0,1,0,0,0),(1,0,0,0,0)]` sind ungleich.

```
classCombinations :: Int -> (Int,Int,Int,Int,Int) -> [[(Int,Int,Int,Int,Int)]]
classCombinations 1 tc = [[tc]]
classCombinations n tc = [ x:y
                         | x <- allSubClasses tc
                         , y <- classCombinations (n - 1) (tc - x)
                         ]
```

Im Gegensatz zu `classCombinations` erzeugt `orderedClassCombinations` alle `n`-elementigen Listen von 5-Tupeln derart, daß die Elemente der Tupel-Listen aufsteigend geordnet sind.

```
orderedClassCombinations :: Int -> (Int,Int,Int,Int,Int) -> [[(Int,Int,Int,Int,Int)]]
orderedClassCombinations 1 tc = [[tc]]
orderedClassCombinations n tc = [ x:y
                                | x <- allSubClasses tc
                                , y@(y':_) <- orderedClassCombinations (n - 1) (tc - x)
                                , x <= y'
                                ]
```

Beispiel 5.1.1 Die Ergebnisse beider Funktionen, bei gleichen Parametern, verdeutlichen den Unterschied:

```
Main> classCombinations 2 (1,2,0,0,0)
[ [(0,0,0,0,0), (1,2,0,0,0)]
, [(0,1,0,0,0), (1,1,0,0,0)]
, [(0,2,0,0,0), (1,0,0,0,0)]
, [(1,0,0,0,0), (0,2,0,0,0)]
, [(1,1,0,0,0), (0,1,0,0,0)]
, [(1,2,0,0,0), (0,0,0,0,0)] ]

Main> orderedClassCombinations 2 (1,2,0,0,0)
[ [(0,0,0,0,0), (1,2,0,0,0)]
, [(0,1,0,0,0), (1,1,0,0,0)]
, [(0,2,0,0,0), (1,0,0,0,0)] ]
```

5.2 Ausschluß bestimmter Ausdrücke

Um die Ausdrucksmengen klein zu halten und nur für die zu testenden Reduktionsdiagramme relevante Ausdrücke zu erzeugen, wollen wir bestimmte Ausdrücke erst gar nicht generieren. Dies sind im einzelnen:

a) *Abstraktionen*
Ein Ausdruck in HNF ist nicht weiter reduzierbar.

b) *Ausdrücke mit mehr als einer Abstraktion innerhalb von Abstraktionen*
Diese Ausdrücke sind für unsere Betrachtungen in erster Linie nicht weiter interessant, da das Auftreten interner Reduktionen innerhalb von Abstraktionen ausgeschlossen wird. Falls natürlich Reduktionen entwickelt werden sollen, die in Kontexten mit mehrmals verschachtelten Abstraktionen Probleme haben könnten, dann muß das im Termgenerator geändert werden.

c) `case`-*Ausdrücke*

- *deren erstes Argument eine Abstraktion ist,*
- *deren erstes Argument eine ungesättigte Konstruktoranwendung ist,* oder
- *die mindestens eine Abstraktionen als Alternative enthalten.*

Diese Ausdrücke sind ebenfalls irreduzibel.

Zu bemerken ist, daß diese Einschränkungen natürlich nur für den Ausdruck gelten, nicht jedoch für seine Teilausdrücke. Z. B. kann eine Alternative eines `case`-Ausdrucks sehr wohl eine Abstraktion sein, wenn der `case`-Ausdruck wiederum auf einen weiteren Teilausdruck angewandt wird.

Der Schalter `toplevel` sorgt dafür, daß keine Abstraktionen an oberster Ebene erzeugt werden, da diese Ausdrücke schon in HNF sind. Er wird bei Ausdrücken, die Abstraktionen enthalten dürfen auf `False` gesetzt. Bei der Generierung von Teilausdrücken wird der Parameter `toplevel` rekursiv weitergereicht:

- an den Generator von E im Term (`let(rec)` ... `in` E),

- an den Generator von E im Term (`case` E `of` $Alts$),
- an die Generatoren der $Alts$ im Term (`case` E `of` $Alts$), sowie
- an beide Generatoren der E_i im Term (`choice` E_1 E_2).

Die folgende Tabelle faßt die vier Parameter noch einmal übersichtlich zusammen:

Parameter	Typ	Verhalten
`toplevel`	Bool	Zeigt an, daß der Term kein Teilterm eines anderen ist, so daß z. B. keine ungesättigten Konstruktoren erzeugt werden.
`carity`	Int	Die aktuelle Sättigung einer Konstruktoranwendung. Bei jeder Applikation wird dieser Wert für den linken Term erhöht, so daß in einem Blatt des Term-Baumes nur Konstruktoren mit passender Stelligkeit stehen können.
`csat`	Bool	Innerhalb eines `case` wird dieser Parameter auf `True` gesetzt, um gesättigte Konstruktoranwendungen zu erhalten.
`ctype`	CType	Dieser Parameter enthält den Typ des `case`-Ausdrucks, der generiert werden soll, so daß auch nur Terme dieses Typs (natürlich nur auf nächster Ebene) erzeugt werden.[4]
`rec`	Bool	Bestimmt den zu verwendenden Kalkül (`True` = rekursiver Kalkül).
`recvar`	VName	Im rekursiven Kalkül können Terme, die an Variablen gebunden sind, die Variable selbst enthalten. Um wiederum offensichtlich nicht reduzierbare Terme zu vermeiden[5], wird der Name der Variable an den rekursiven Aufruf des Termgenerators weitergegeben.
`maxb`	Int	Gibt an wieviele Bindungen ein `letrec`-Ausdruck enthalten darf. Standardmäßig erlauben wir zwei Bindungen, da dadurch verschränkt rekursive Terme erzeugt werden können. Der Standardwert läßt sich ändern.

5.3 Verwendung der generierten Terme

5.3.1 Korrektur der Variablennamen

Bedingt durch die parallele Erzeugung der Ausdrucksmengen enthalten die Ausdrücke noch Variablen mit gleichen Namen. Unserer Konvention (siehe Konvention 3.1.1) folgend müssen die Variablen so umbenannt werden, daß jede gebundene Variable einen eindeutigen Namen erhält. Die Korrektur der Variablennamen übernimmt die Funktion `correctCoreExpr` (siehe Anhang A.1).

[4] Da wir kein Typsystem implementiert haben, ist das ein Trick, um die Erzeugung pathologischer Terme zu verringern.

[5] In erster Linie werden Terme wie `letrec x=x in x` verhindert. Diese Terme vergrößern die Termmenge nicht unerheblich, wenn sie im Term-Graphen „ganz unten“ auftreten.

5.3.2 Einsetzen generierter Terme in wählbare Ausdrücke

Als Super-Term kann ein beliebiger Ausdruck gewählt werden, der ein Loch (`[]`) enthält. Intern verwenden wir zur Kennzeichnung eines Lochs eine „Variable ohne Namen“, also `EVar Unmarked ""`. Der Aufruf `replaceVar "" g e` setzt den Ausdruck `g` in das Loch (`""`) des Ausdrucks `e` ein[6].

Z. B. kann der Ausdruck `case [] of Zero -> True, Succ x -> False` mit den generierten Termen „befüllt“ werden. Dadurch lassen sich bestimmte Termstrukturen gezielt untersuchen, und die Anzahl der zu prüfenden Terme herabsetzen.

5.3.3 Größe der Ausdrucksmengen

Zum Zählen der generierten Ausdrücke dienen die Funktionen `countCoreExprs` und `cCoreExprs`. Sie sind genauso aufgebaut wie `generateCoreExprs` und `genCoreExprs`, mit dem Unterschied, daß das Ergebnis nicht eine Liste von Ausdrücken ist, sondern eben deren Anzahl. Da hier keine Listenelemente verwaltet werden müssen, sondern nur `Integer`-Werte, ist die Berechnung der Mengengröße auch sehr viel schneller. Auf die Entwicklung einer Rekursionsformel wurde wegen der komplizierten Parameterstruktur verzichtet.

Aus Platzgründen ist die Funktion `cCoreExprs` hier nicht abgebildet, da sie prinzipiell wie `genCoreExprs` arbeitet.

```
countCoreExprs rec maxb classonly termClass varsAndCons
    = if classonly
         then cCoreExprs termClass varsAndCons settings
         else sum [ cCoreExprs termclass varsAndCons settings
                  | termclass <- allSubClasses termClass
                  ]
         where
           settings = (True, 0, True, "", rec, "", maxb)
```

[6] Falls mehrere Löcher im Term vorhanden sind, dann wird der Ausdruck in jedes Loch eingesetzt. Das birgt jedoch die Gefahr, bestimmte Termklassen zu „überspringen“.

Exemplarisch soll die folgende Tabelle zeigen, wie sich die Größen der generierten Termklassen unterscheiden:

Termklasse	Typen	Anzahl	Anzahl (`letrec`)
(1,2,2,1,0)	Bool	7.295.150	7.255.454.439
	Nat	7.460.231	7.815.984.848
	List	8.566.904	9.270.392.035
	alle	2.453.994.335	1.925.596.067.705
(1,2,2,1,1)	Bool	397.809.753	649.329.772.778
	Nat	377.214.073	692.976.440.865
	List	473.731.386	858.334.812.395
	alle	248.465.966.877	288.471.078.484.019
(2,2,2,2,0)	Bool	15.839.229.912	30.728.107.306.278
	Nat	16.955.654.855	36.007.759.324.024
	List	32.738.831.535	59.269.897.116.106
	alle	26.526.086.738.541	40.292.941.362.008.944

Die immens großen Unterschiede bei Verwendung aller Datentypen rühren daher, daß wir keine Typ-Überprüfung in die Termgenerierung eingebaut haben, weshalb eine große Anzahl der Terme nicht wohlgetypt sind. Durch die oben besprochenen lokalen Optimierungen konnte die Anzahl jedoch etwas verringert werden.

Allein die Zeit zum Zählen der 40 Billiarden Terme beträgt ca. eine Stunde. Der verwendete Rechner ist ein Pentium II mit 266 MHz, 128 MB RAM, Linux 2.2, JONAH mit GHC 4.06 kompiliert.

Kapitel 6

Validierung

In diesem Kapitel stellen wir die Validierung von Reduktionsdiagrammen vor. Abbildung 6.1 veranschaulicht den Ablauf der Validierung. In den vorhergehenden Kapiteln haben wir die Kernsprache und den Termgenerator beschrieben, die zusammen oder getrennt voneinander eine Menge von Termen liefern, die für die Validierung der Diagrammsätze benutzt wird. Wir werden nun den Aufbau der Regeldatei (sie enthält die Diagrammdefinitionen), den Parser dieser Datei und den Regel- bzw. Diagrammgenerator genauer beschreiben.

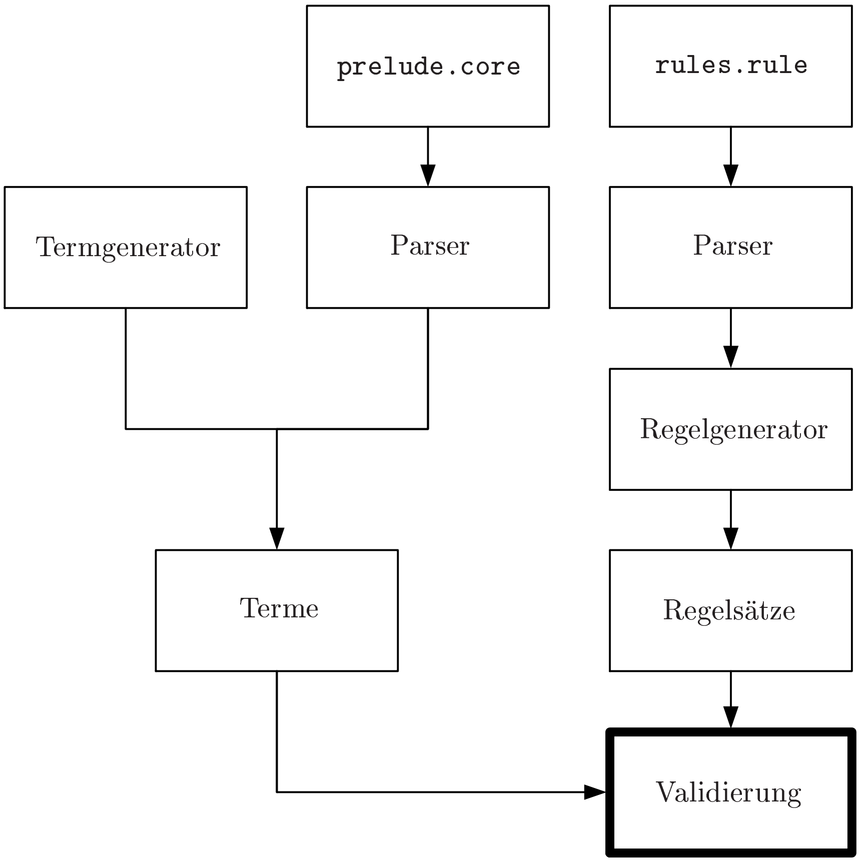

Abbildung 6.1: Ablauf der Validierung

6.1 Darstellung der Reduktionsdiagramme

Um die vorgestellten Kalküle mit ihren Reduktions-Regeln und Erweiterungen zu validieren, benötigen wir eine geeignete Darstellung für Reduktionsdiagramme, so daß unser Programm sie weiterverarbeiten kann. Wir definieren eine Sprache für Reduktionsdiagramme und zeigen, wie diese Reduktionsdiagramme für eine geeignete Menge an Termen validiert werden können.

Die in Kapitel 4.2 eingeführten Reduktionsdiagramme und Reduktionsdiagramm-Sätze werden in einer sogenannten Regel-Datei[1] definiert, standardmäßig verwenden wir als Namen „`rules.rule`“. Die dafür benutzte Sprache lehnt sich sehr an den von uns gewählten Formalismus für Vertauschungsdiagramme an.

6.1.1 Aufbau der Regeldatei

Die Syntax der Regeldatei ist in Abbildung 6.2 zu sehen. Die Regeln werden zu Sätzen von Regeln zusammengefaßt, die mit einem (beschreibenden) Namen versehen werden. Eine Regel besteht aus zwei durch ein Pfeilsymbol (`==>`) getrennten Teilen — der *linken* und der *rechten* Seite. Die linke Seite bestimmt die Reduktionsfolge, die durch die Folge auf der rechten Seite ersetzt werden kann.

Eine Reduktionsfolge besteht aus mindestens einem Redextyp, wobei auf der rechten Seite einer Regel auch kein Reduktionstyp stehen darf. Wird keine explizite Reduktionsart (Normalordnung oder interner Reduktionstyp) angegeben, so wird ein regulärer Reduktionstyp angenommen.

Zur Verallgemeinerung von Reduktionsregeln kann, statt eines bestimmten Reduktionstyp, eine Variable angegeben werden, die durch Reduktionstypen einer definierten Menge belegt werden kann. Weiterhin führen wir Wiederholungs- und Mengenkonstrukte ein, die es erlauben komplexe Reduktionsdiagramme zu definieren.

6.1.2 Datenstrukturen für Regeln

Zur Implementierung der Diagramme benötigen wir den abstrakten Datentyp `Reduction`, der eine Reduktion bzw. einen Reduktionstyp als NO-, interne oder reguläre Reduktion spezifiziert. Zusätzlich wird mit dem Datentyp der Mengenkonstrukt (siehe Kapitel 6.1.7) realisiert: der rekursive Konstruktor `SET` beinhaltet eine Menge von Reduktionen und eine untere und obere Schranke, die die Länge der zu erzeugenden Reduktionsfolge einschränken.

[1] Da jedes Reduktionsdiagramm eine Regel zum Umstellen einer Reduktionsfolge darstellt, verwenden wir synonym auch den Begriff *Reduktions-Regel* oder einfach nur *Regel*. Um eine Verwechslung mit den Reduktions-Regeln eines Kalküls zu vermeiden, benutzen wir beide Begriffe im jeweiligen Kontext.

<table>
<tr><td>rules</td><td>→</td><td>$ruleset_1 \ldots ruleset_n$</td><td>$n \geq 1$</td></tr>
<tr><td>ruleset</td><td>→</td><td><code>></code> var <code>:</code> $rule_1 \ldots rule_n$</td><td>$n \geq 1$</td></tr>
<tr><td>rule</td><td>→</td><td><code>></code> $redgroup_1 \ldots redgroup_m$</td><td>$m \geq 1$</td></tr>
<tr><td></td><td></td><td><code>==></code> $redgroup_1 \ldots redgroup_n$</td><td>$n \geq 0$</td></tr>
<tr><td></td><td></td><td>[<code>where</code> $vartype_1 \ldots vartype_k$]</td><td>$k \geq 1$</td></tr>
<tr><td>vartype</td><td>→</td><td>var <code><-</code> <code>{</code> $redtype_1, \ldots, redtype_n$<code>}</code></td><td>$n \geq 1$</td></tr>
<tr><td>redgroup</td><td>→</td><td>redv</td><td></td></tr>
<tr><td></td><td>|</td><td><code>(</code> $redv_1 \ldots redv_n$ <code>)</code> rep</td><td>$n \geq 1$ (Gruppe)</td></tr>
<tr><td></td><td>|</td><td><code>{</code> $red_1, \ldots, red_n$ <code>}</code> [<code>*</code> | <code>+</code> | <code>?</code>]</td><td>$n \geq 1$ (Menge)</td></tr>
<tr><td>red</td><td>→</td><td>[redkind <code>.</code>] redtype</td><td>Reduktion</td></tr>
<tr><td>redv</td><td>→</td><td>red</td><td></td></tr>
<tr><td></td><td>|</td><td>[redkind <code>.</code>] var</td><td>Variable</td></tr>
<tr><td>rep</td><td>→</td><td><code>*</code> | <code>+</code> | <code>?</code></td><td>Wiederholungen</td></tr>
<tr><td></td><td>|</td><td>var [<code>*</code> | <code>+</code> | <code>?</code>]</td><td></td></tr>
<tr><td>redkind</td><td>→</td><td><code>n</code> | <code>N</code> | <code>no</code> | <code>NO</code></td><td>Normalordnung</td></tr>
<tr><td></td><td>|</td><td><code>i</code> | <code>I</code></td><td>Intern</td></tr>
<tr><td></td><td>|</td><td><code>r</code> | <code>R</code></td><td>Regulär</td></tr>
<tr><td>redtype</td><td>→</td><td><code>lbeta</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>lapp</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>llet</code> | <code>lletr1</code>[1)] | <code>lletr2</code>[1)]</td><td></td></tr>
<tr><td></td><td>|</td><td><code>abs</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>cp</code> | <code>cpd</code>[2)] | <code>cpt</code>[2)]</td><td></td></tr>
<tr><td></td><td>|</td><td><code>case</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>lcase</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>ldel</code>[2)] | <code>lcom</code>[2)] | <code>lcvt</code>[2)] | <code>lcvd</code>[2)] | <code>eql</code>[2)] | <code>ucp</code>[2)]</td><td></td></tr>
<tr><td></td><td>|</td><td>name</td><td></td></tr>
<tr><td>name</td><td>→</td><td>capital $char_1 \ldots char_n$</td><td>$n \geq 0$</td></tr>
<tr><td>var</td><td>→</td><td>alpha $char_1 \ldots char_n$</td><td>$n \geq 0$</td></tr>
<tr><td>char</td><td>→</td><td>alpha | capital | digit | special</td><td></td></tr>
<tr><td>alpha</td><td>→</td><td><code>a</code> | ... | <code>z</code></td><td></td></tr>
<tr><td>capital</td><td>→</td><td><code>A</code> | ... | <code>Z</code></td><td></td></tr>
<tr><td>digit</td><td>→</td><td><code>0</code> | ... | <code>9</code></td><td></td></tr>
<tr><td>special</td><td>→</td><td><code>_</code> | <code>-</code> | <code>/</code></td><td></td></tr>
</table>

1) Reduktionstypen des $\lambda_{nd,rec}$–Kalküls.

2) Erweiterungen der Basiskalküle.

Abbildung 6.2: Die Syntax der Regeldatei

```
data Reduction = NO  ReductionType
               | I   ReductionType
               | R   ReductionType
               | SET [Reduction] Int Int
               deriving (Eq)
```

Die Reduktionstypen werden wie folgt definiert:

```
data ReductionType
     = VAR      String
     | LBeta
     | LApp
     | LLet
     | LLetrec1
     | LLetrec2 String
     | Abs
     | Copy     String
     | Case
     | CaseLet
     | ND
     | NDL
     | NDR
```

Erweiterte Reduktionstypen sind:

```
     | CopyD    String
     | CopyT    String
     | LDel     String
     | LDelS    String
     | LDelCyc1 [String]
     | LDelCyc2
     | LDup     String
     | LCVT     String
     | LCVD     String
     | LCom
     | UCP      String
     | EQL      String
     deriving (Ord)
```

Zu beachten ist, daß wir bestimmte Reduktionstypen genauer spezifizieren. Alle in dieser Arbeit vorgestellten Reduktionstypen sind durch einen Variablennamen erweitert. Dieser Trick erlaubt es, komplizierte Kopierregeln wie (*cpt*) einfacher zu reduzieren, da der markierte Redex nicht noch einmal analysiert werden muß (siehe Anhang A).

Die in der Regeldatei definierten Regeln werden zuerst in ein Zwischenformat gebracht, aus dem dann vor dem Validieren die eigentlichen Regeln (Diagramme) erzeugt werden. Eine `ParsedReduction` ist entweder eine einzelne Reduktion oder eine Folge von Reduktionen, die wiederholt werden können, wobei `Generator` die Wiederholungen durch eine Variable und ein Wiederholungszeichen spezifiziert.

```
type Generator       = (String, Char)

type ParsedReduction = ([Reduction], Generator)
```

Nach dem Parsen einer Regel erhalten wir ein Tripel mit den Reduktionen der linken und rechten Seite und einer optionalen Menge von Variablenzuweisungen (`ReductionSet`). Ein Regelsatz ist eine Liste von Regeln, die durch einen Namen gekennzeichnet ist.

```
type ReductionSet    = (String, [ReductionType])

type ParsedRule = ([ParsedReduction], [ParsedReduction], [ReductionSet])

type ParsedRuleSet = (String, [ParsedRule])
```

Wenn die Regeln aus dem Zwischenformat in das eigentliche Regelformat übersetzt worden sind, dann erhalten wir den Typ `Rule` und für Regelsätze `RuleSet`.

```
type Rule = ([Reduction], [Reduction])

type RuleSet = (String, [Rule])
```

6.1.3 Implementierung des Parsers

Da der benutzte Parser-Generator schon im Zusammenhang mit der Kernsprache ausführlich beschrieben wurde, soll hier nur kurz auf die Implementierung des Parsers für Reduktions-Regeln eingegangen werden.

Die Token `ddash` symbolisieren die Regel-Implikation `==>` und `from` stellt den Pfeil `<-` für die Variablenzuweisung dar.

```
%name parser
%tokentype { Token }
%token VarId        { (LexemVar   $$  , _, _) }
       Name         { (LexemVar   $$  , _, _) }
       Kind         { (LexemVar   $$  , _, _) }
       '>'          { (LexemGT        , _, _) }
       '.'          { (LexemDot       , _, _) }
       ','          { (LexemComma     , _, _) }
       ':'          { (LexemColon     , _, _) }
       '*'          { (LexemAsterisk  , _, _) }
       '+'          { (LexemPlus      , _, _) }
       '?'          { (LexemQuestion  , _, _) }
       '('          { (LexemLBracket  , _, _) }
       ')'          { (LexemRBracket  , _, _) }
       '{'          { (LexemLBracket1 , _, _) }
       '}'          { (LexemRBracket1 , _, _) }
       ddash        { (LexemDdash     , _, _) }
       from         { (LexemLarrow    , _, _) }
       where        { (LexemWhere     , _, _) }
```

Alle Regelsätze werden zu einem Regel-Pool zusammengefaßt, die Regelsätze sind durch einen eindeutigen Namen unterscheidbar.

```
RulePool :: { [ParsedRuleSet] }
          : RuleSet RulePool        { $1 : $2 }
          ! RuleSet                 { [$1]    }

RuleSet  :: { ParsedRuleSet }
          : '>' Name ':' Rules      { ($2, $4) }
```

Ein Regelsatz enthält eine oder mehrere Regeln. Jede Regel kann eine „leere“ rechte Seite und eine oder mehrere Variablenzuweisungen besitzen.

```
Rules    :: { [ParsedRule] }
          : Rule Rules              { $1 : $2 }
          ! Rule                    { [$1]    }

Rule     :: { ParsedRule }
          : '>' RedG ddash                          { ($2, [], []) }
          ! '>' RedG ddash      where TypeVars  { ($2, [], $5) }
          ! '>' RedG ddash RedG                     { ($2, $4, []) }
          ! '>' RedG ddash RedG where TypeVars  { ($2, $4, $6) }
```

Eine Reduktionsfolge besteht aus einer oder mehreren Reduktionen, die wiederum als wiederholte Folge oder als Mengenkonstrukt auftreten können. Bei Wiederholungen kann eine Variable angegeben werden, um Folgen zu synchronisieren (siehe Kapitel 6.1.6, Seite 84).

```
RedG     ::  [ParsedReduction]
          : Reds                     ($1, ("",'1')) : []
          ! Reds RedG                ($1, ("",'1')) : $2
          ! '(' Reds  ')' Gen        ($2, $4) : []
          ! '(' Reds  ')' Gen RedG   ($2, $4) : $5
          ! '' RedsC ''             ([SET $2 1 1],("",'1')) : []
          ! '' RedsC '' '*'         ([SET $2 0 0],("",'1')) : []
          ! '' RedsC '' '+'         ([SET $2 1 0],("",'1')) : []
          ! '' RedsC '' '?'         ([SET $2 0 1],("",'1')) : []
          ! '' RedsC ''     RedG    ([SET $2 1 1],("",'1')) : $4
          ! '' RedsC '' '*' RedG    ([SET $2 0 0],("",'1')) : $5
          ! '' RedsC '' '+' RedG    ([SET $2 1 0],("",'1')) : $5
          ! '' RedsC '' '?' RedG    ([SET $2 0 1],("",'1')) : $5

Gen      ::  Generator
          : '*'                      ("",'*')
          ! '+'                      ("",'+')
          ! '?'                      ("",'?')
          ! VarId                    ($1,'*')
          ! VarId '*'                ($1,'*')
          ! VarId '+'                ($1,'+')
          ! VarId '?'                ($1,'?')
```

Statt einer bestimmten Reduktion können auch Variablen angegeben werden, so daß das Token `RedV` beide Möglichkeiten berücksichtigen muß. `Reds` ist eine Folge dieser Token, `RedsC` ist eine durch Kommata getrennte Liste von Reduktionen (ohne Variablen).

```
Reds     :: { [Reduction] }
          : RedV Reds                 { $1 : $2 }
          | RedV                      { [$1]    }

RedsC   :: { [Reduction] }
          : Red ',' RedsC             { $1 : $3 }
          | Red                       { [$1]    }

Red      :: { Reduction }
          : Kind '.' VarId            { makeReduction $1  $3 }
          | VarId                     { makeReduction "r" $1 }

RedV     :: { Reduction }
          : Red                       { $1 }
          | Kind '.' VarId            { makeReductionOrVar $1  $3 }
          | VarId                     { makeReductionOrVar "r" $1 }
```

Wenn in einer Regel Variablen definiert sind, so müssen diese bei der Umwandlung in das zur Validierung benutzte Regelformat mit Reduktionen belegt werden. Alle möglichen Reduktionstypen[2] werden in geschweiften Klammern der Variablen zugewiesen. Die Variablenzuweisungen werden in einer Liste gesammelt (`ReductionSet`).

```
TypeVars :: { [ReductionSet] }
          : TypeVar TypeVars          { $1 : $2  }
          | TypeVar                   { [$1]     }

TypeVar  :: { ReductionSet }
          : VarId from '{' Types '}' { ($1, $4) }

Types    :: { [ReductionType] }
          : VarId ',' Types           { (decodeReductionType $1) : $3 }
          | VarId                     { [decodeReductionType $1]      }
```

Anhand der folgenden Beispiele soll gezeigt werden, welche Möglichkeiten der Charakterisierung von Regeln (Diagrammen) unsere Regel-Sprache bietet.

6.1.4 Einfache Reduktionsdiagramme

Ein Reduktionsdiagramm könnte z. B. sein:

$$(i, llet) \circ (no, cp) \quad \leadsto \quad (no, cp) \circ (i, llet)$$

Diese Regel wird analog in der Regeldatei definiert:

```
> i.llet n.cp ==> n.cp i.llet
```

[2]Weil die Variable `a` in der Regel bereits charakterisiert ist (`no.a`, `i.a`, oder `r.a`), dürfen hier nur die Reduktionstypen bestimmt werden.

6.1.5 Reduktionsdiagramme mit Variablen

Die meisten im Rahmen dieser Arbeit validierten Reduktionsdiagramme können verallgemeinert werden. D. h. die oben angegebene Regel gilt nicht nur für (no, cp), sondern auch für $(no, lapp)$ u. a., so daß sie folgendermaßen umformuliert werden kann:

$$(i, llet) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, llet), \qquad a \in \{cp, lapp\}$$

Diese Regel wird analog in der Regeldatei definiert:

```
> i.llet n.a ==> n.a i.llet where a <- {cp,lapp}
```

Da wir die Sprache der Regeldatei so gewählt haben, daß jede Regel mit einem `>`-Zeichen eingeleitet wird, ist es möglich, die Regel über mehrere Zeilen zu verteilen, und so übersichtlicher darzustellen.

```
> i.llet n.a ==> n.a i.llet
  where
    a <- {cp,lapp}
```

Diese allgemeine Darstellung der Regel muß vor der Validierung noch in eine Menge von spezialisierten Regeln umgewandelt werden. Die Funktion `instantiateRules` erzeugt aus Regeln mit Variablen eine Menge von Instanzen mit allen Möglichkeiten der Variablenbelegung. Zum Beispiel:

```
instantiateRules 0 "n.a ==> n.b where a <- {cp,llet} b <- {lapp,lbeta}"
```

liefert:

```
> n.cp   ==> n.lapp
> n.llet ==> n.lapp
> n.cp   ==> n.lbeta
> n.llet ==> n.lbeta
```

Der erste Parameter der Funktion (vom Typ `Int`) wird innerhalb von `instantiateRules` an die Funktion `generateRules` weitergegeben und gibt die obere Schranke für die Erzeugung von Wiederholungen an, die in diesem Beispiel aber nicht benötigt wird.

6.1.6 Reduktionsdiagramme mit Wiederholungen

Teilfolgen von Reduktionen eines Reduktionsdiagramms (auch mit Variablen) können mit Wiederholungsanweisungen versehen werden. Die Funktion `generateRules` erzeugt aus einer solchen Regel Instanzen mit einer vorher festgelegten oberen Grenze. Die Anweisungen hinter den geklammerten Reduktionen verhalten sich ähnlich denen der *regulären Ausdrücke*. Aus den Kombinationen der Werte jeder Variablen ergeben sich die Möglichkeiten der zu erzeugenden Regeln.

Es ist möglich, die Längen der beiden Reduktionsfolgen eines Reduktionsdiagramms zu synchronisieren, indem auf beiden Seiten eine Variable mit gleichem Namen verwendet wird. Die Zeichen `*`, `+` und `?` geben an, welche Werte die Variablen annehmen sollen:

- `(...)i` bzw. `(...)i*` bedeutet, daß die geklammerte Folge *keinmal* oder bis zu `maxGen`-mal wiederholt wird.
- `(...)i+` wiederholt den geklammerten Teil mindestens *einmal* und maximal `maxGen`-mal.
- `(...)i?` bedeutet, daß der geklammerte Teil *einmal* oder *gar nicht* auftritt.

`maxGen` wird der Funktion `generateRules` übergeben.

Anhand dreier Beispiele soll diese Art der Regel-Definition verdeutlicht werden.

Synchronisation

`generateRules 3 "r.Red1 (r.Red2)i+ ==> (r.Red3)i+"`
liefert die Instanzen:

```
r.Red1 r.Red2               ==> r.Red3                (i = 1)
r.Red1 r.Red2 r.Red2        ==> r.Red3 r.Red3         (i = 2)
r.Red1 r.Red2 r.Red2 r.Red2 ==> r.Red3 r.Red3 r.Red3  (i = 3)
```

Verschiedene Variablen

`generateRules 2 "r.Red1 (r.Red2)i ==> (r.Red3)k+"`
bzw.
`generateRules 2 "r.Red1 (r.Red2)* ==> (r.Red3)+"`
liefert die Instanzen:

```
r.Red1               ==> r.Red3         (i = 0, k = 1)
r.Red1 r.Red2        ==> r.Red3         (i = 1, k = 1)
r.Red1 r.Red2 r.Red2 ==> r.Red3         (i = 2, k = 1)
r.Red1               ==> r.Red3 r.Red3  (i = 0, k = 2)
r.Red1 r.Red2        ==> r.Red3 r.Red3  (i = 1, k = 2)
r.Red1 r.Red2 r.Red2 ==> r.Red3 r.Red3  (i = 2, k = 2)
```

Wiederholung längerer Folgen

`generateRules 3 "(r.Red1 r.Red2)i+ ==> r.Red3"`
bzw.
`generateRules 3 "(r.Red1 r.Red2)+  ==> r.Red3"`
liefern die Instanzen:

```
r.Red1 r.Red2                             ==> r.Red3  (i = 1)
r.Red1 r.Red2 r.Red1 r.Red2               ==> r.Red3  (i = 2)
r.Red1 r.Red2 r.Red1 r.Red2 r.Red1 r.Red2 ==> r.Red3  (i = 3)
```

6.1.7 Reduktionsdiagramme mit Mengenkonstrukt

Bei einigen Regeln ist nicht immer klar, wie oft und in welcher Reihenfolge bestimmte Reduktionstypen auftreten können. Wenn man nun alle Möglichkeiten, die entstehen können, zu Regeln formuliert, dann erhält man eine große, unübersichtliche Menge von Diagrammen. Aus diesem Grund kann eine Regel einen Mengenkonstrukt enthalten, der genau diese Möglichkeiten abdeckt.

Betrachten wir die Regel
`> i.llet {n.lapp, n.llet, n.lcase}+ ==> {n.lapp, n.llet, n.lcase}+ i.llet`.

Der Mengenkonstrukt `{n.lapp, n.llet, n.lcase}+` stellt alle Reduktionfolgen der Länge 1 bis `maxGen` dar, die aus den angegebenen Redexen gebildet werden können.

Entgegen der Syntax bei Wiederholungsanweisungen, kann hier nicht mittels Variablen synchronisiert werden, und es sind nur die Konstrukte `{` ... `}+` und `{` ... `}*` möglich. Innerhalb der Mengenklammer dürfen weder variable Reduktionstypen noch Wiederholungsanweisungen auftreten.

Variable Reduktionstypen und Wiederholungen werden vor der eigentlichen Validierung aufgelöst, so daß aus den (allgemeinen) Regeln eine endliche Menge von Diagrammen erzeugt wird, die zur Validierung benutzt wird (siehe oben). Da die Anzahl der erzeugten Regeln beim Mengenkonstrukt erheblich größer ist, werden diese Regeln direkt bei der Validierung *simuliert* (siehe Kapitel 6.3.1).

Dadurch bleibt die Größe des Diagrammsatzes zwar klein, es besteht aber nicht mehr die Möglichkeit einer Synchronisation von Reduktionsfolgen. Vorteilhaft ist jedoch, daß bei der Simulation unmögliche Reduktionsfolgen schon frühzeitig ausgeschlossen werden können und somit nicht validiert werden müssen.

6.2 Test der einzelnen Reduktionsdiagramme

Die in dieser Arbeit verifizierten Reduktionsdiagramme haben alle eine ähnliche Struktur:

- Die linke Seite beginnt mit einer *internen Reduktion*,
- darauf folgen nur noch *NO-Reduktionen* oder keine Reduktion,
- die rechte Seite enthält *interne* und *NO-Reduktionen*, wobei einer *internen Reduktion* keine *NO-Reduktionen* mehr folgen.

Die interne Reduktion im rechten Teil des Diagramms ist meist vom gleichen Typ wie die erste Reduktion auf der linken Seite. Dies bedeutet allerdings nicht, daß es sich auch um denselben Redex handeln muß, nur der Reduktionstyp ist wichtig.

Die Validierung eines Reduktionsdiagramms wird durch den folgenden Algorithmus beschrieben. Zum Verständnis des Verfahrens gehen wir davon aus, daß kein Mengenkonstrukt innerhalb des Diagramms (Meta-Regel) verwendet wird.

Definition 6.2.1 Sei (i, a) die zu überprüfende interne Reduktion. In einem Ausdruck können mehrere interne Redexe vom Typ a vorkommen. Die *interne Reduktion* (i, a) kann wiederum mehrere Redukte liefern. Für jeden dieser Redexe und jedes Redukt muß ein Diagramm $\mathcal{D}$ im Diagrammsatz $\mathcal{S}_{\mathcal{D}}$ enthalten sein, durch das der Term t geschlossen werden kann. Das Diagramm $\mathcal{D}$ (siehe Abbildung 6.3, Seite 88)[3] sei gegeben durch:

$$(i, a) \circ (no, r_1) \circ \cdots \circ (no, r_n) \quad \rightsquigarrow \quad (no, r'_1) \circ \cdots \circ (no, r'_m) \circ (i, r''_1) \circ \cdots \circ (i, r''_l)$$

$$\text{mit } n, l \geq 0 \text{ und } m \geq 1$$

1. Wir reduzieren den Ausdruck t gemäß der angegebenen Reduktionsfolge der rechten Regelseite. Das Ergebnis ist eine Menge von Ausdrücken $T = \{t'_1, \ldots, t'_i\}$ mit $i \geq 1$, da die Reduktionsfolge der rechten Regelseite mit internen Reduktionen enden kann.

2. $I = \{s \mid t \xrightarrow{(i,a)} s\}$ ist die Menge aller Redukte aller internen Reduktionen vom Typ a im Term t.

3. Für alle $s \in I$ bearbeite Schritt 3.1 und 3.2:

   3.1 s' ist das Ergebnis der Reduktion $s \xrightarrow{no,r_1} \ldots \xrightarrow{no,r_n} s'$.

   3.2 Wenn s' in der Ergebnismenge T enthalten ist, dann kann t mittels des Diagramms $\mathcal{D}$ geschlossen werden.

4. Können alle $s \in I$ geschlossen werden, dann sind die Diagramme des Diagrammsatzes $\mathcal{S}_{\mathcal{D}}$ erfolgreich validiert.

Bei Verwendung des Mengenkonstrukts wird die Validierung etwas komplizierter, da die Reihenfolge der einzelnen Reduktionen und die Längen der möglichen Reduktionsfolgen nicht bekannt sind.

Definition 6.2.2 (Algorithmus *Validate*) Sei (i, a) die zu überprüfende interne Reduktion. Das Diagramm $\mathcal{D}$ habe o.b.d.A. die Form:

$$(i, a) \circ \{(no, r_1), \ldots, (no, r_n)\}^* \quad \rightsquigarrow \quad \{(no, r'_1), \ldots, (no, r'_m)\}^* \circ (i, b)$$

$$\text{mit } n, m \geq 1$$

1. Wir reduzieren den Ausdruck t gemäß der angegebenen Reduktionsfolge der rechten Regelseite, wobei alle möglichen Reduktionsfolgen des Ausdrucks $\{(no, r_1), \ldots, (no, r_m)\}^*$ Verwendung finden. Das Ergebnis ist eine Menge von Ausdrücken $T = \{t'_1, \ldots, t'_i\}$ mit $i \geq 1$.

2. $I = \{s \mid t \xrightarrow{(i,a)} s\}$ ist die Menge aller Redukte aller internen Reduktionen vom Typ a im Term t.

[3] Im Zusammenhang mit Meta-Regeln bzw. (Reduktions-)Diagrammen sprechen wir oft von der *linken* und *rechten Seite* einer Regel. Dies ist jedoch in der Abbildung genau umgekehrt dargestellt. D. h. die linke Seite der Abbildung stellt die rechte Seite der Regel dar, und der grau unterlegte Teil die linke Regelseite.

3. Für alle $s \in I$ bearbeite Schritt 3.1 und 3.2:

   3.1 $S = \{s' \mid s \xrightarrow{\{(no,r_1),\ldots,(no,r_n)\}^*} s'\}$, also die Ergebismenge aller möglichen Reduktionsfolgen der angegebenen Reduktionen-Menge.

   3.2 Wenn mindestens ein $s' \in S$ in der Ergebnismenge T enthalten ist (bzw. $S \cap T \neq \emptyset$ gilt), dann kann t mittels des Diagramms $\mathcal{D}$ geschlossen werden.

4. Können alle $s \in I$ geschlossen werden, dann sind die Diagramme des Diagrammsatzes $\mathcal{S}_\mathcal{D}$ erfolgreich validiert.

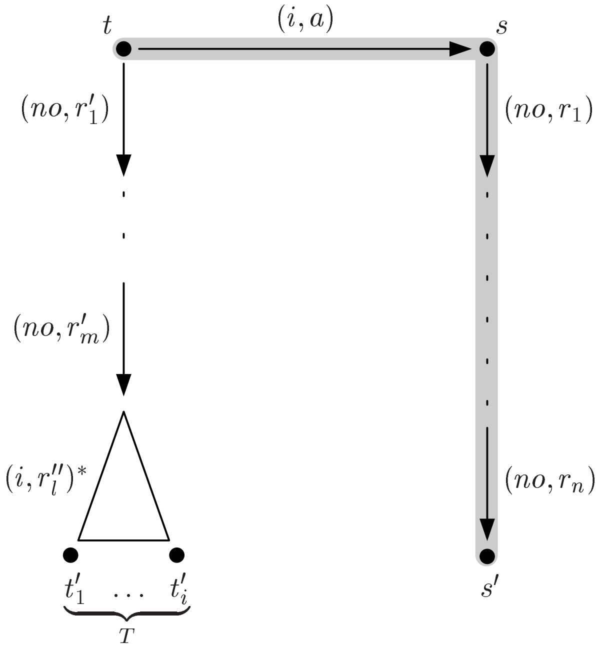

Abbildung 6.3: Validierung eines Reduktionsdiagramms

Im folgenden Kapitel werden wir die Implementierung der Validierung genauer beschreiben.

6.3 Validierung von Reduktionsdiagrammen

Wir definieren für das Ergebnis der Validierung eines Diagramms den Datentyp `RuleMatch`[4], der angibt, ob ein Ausdruck durch ein Diagramm des Diagrammsatzes geschlossen werden kann oder nicht. Im positiven Fall wird das gefundene Diagramm (bzw. Meta-Regel) zurückgegeben.

```
data RuleMatch = RuleNotFound
               | RulesFound [Rule]
               deriving (Show, Eq)
```

[4]Wir hätten auch den Datentyp `Maybe` aus der Haskell-Prelude verwenden können, jedoch wegen der besseren Lesbarkeit des Programm-Codes diesen Weg gewählt.

Das Herz des Validierungsverfahrens bildet die Funktion `closeExpression`. Sie prüft, ob ein Term durch Diagramme eines Diagrammsatzes geschlossen werden kann. Zur Spezifikation des Kalküls dient wiederum der Parameter `rec`, `startReds`[5] ist die Menge der zu validierenden internen Reduktionen (i, a), und `rules` enthält alle Regeln des Diagrammsatzes. Als Ergebnis wird entweder `RuleNotFound` oder die gefundenen Regeln (`RulesFound`) zurückgegeben, wobei Duplikate entfernt werden (`nub`).

Wie oben beschrieben ist der erste Reduktionstyp der linken Diagrammseite eine interne Reduktion. Für alle möglichen Belegungen bzw. alle möglichen Redexe dieses ersten internen Reduktionstyps (`lexpr`) muß das Diagramm geschlossen werden. D. h. wir simulieren alle möglichen Reduktionen beginnend mit allen möglichen internen Redexen des ersten Reduktionstyps.

Zur Optimierung der Validierung werden nur Diagramme des Diagrammsatzes benutzt, die mit der gerade analysierten internen Reduktion beginnen (`red == i`) und mindestens einen NO-Reduktionstyp enthalten (`any isNOreduction lhs || any isNOreduction rhs`).

```
closeExpression :: Bool -> [Reduction] -> [Rule] -> CoreExpr -> RuleMatch
closeExpression rec startReds rules expr
    = case [ take 1 [ rule
                    | rule@(i:lhs,rhs) <- rules
                    , red == i
                    ,    any isNOreduction lhs
                      || any isNOreduction rhs
                    , tryRule rec (lhs, rhs) (lexpr,expr)
                    ]
           | red   <- startReds
           , lexpr <- concat [ reduceCoreExpr rec 1 e
                             | e <- markReduction rec red expr
                             ]
           ]
        of [] -> RuleNotFound
           xs -> if [] `elem` xs
                    then RuleNotFound
                    else RulesFound (nub $ concat xs)
```

Die Funktion `tryRule` simuliert beide Seiten der Regel, wobei als Start-Term für die linke Seite das Ergebnis der ersten internen Reduktion übergeben wird. Im Vergleich zu Abbildung 6.3 ist `lexpr` der Term s und `rexpr` der Term t. Die rechte Seite des Diagramms enthält nur Reduktionen, die existenzquantifiziert sind. Deshalb reicht es, wenn nur *ein* Ausdruck aus der linken Simulation[6] (S) in der Menge der rechten Simulation (T) enthalten ist, bzw. $T \cap S \neq \emptyset$ gilt.

[5] Die (cp)-Reduktion ist aufgeteilt in eine (cpt)- und (cpd)-Reduktion, so daß auch beide Reduktionen überprüft werden müssen.

[6] Weil die linke Seite schon einmal reduziert worden ist, wird hier mit *Level* 2 fortgefahren.

```
tryRule :: Bool -> Rule -> (CoreExpr, CoreExpr) -> Bool
tryRule rec (lhs, rhs) (lexpr,rexpr)
    = [] /= intersect les res
        where
          les = simulateReduction rec 2 lhs [lexpr]
          res = simulateReduction rec 1 rhs [rexpr]
```

Die Simulation einer gegebenen Folge von Reduktionstypen ist wegen der gleichzeitigen Simulation des Mengenkonstrukts etwas kompliziert, so daß wir zuerst eine „einfachere“ Version der Funktion `simulateReduction` betrachten wollen.

6.3.1 Simulation einer Reduktionsfolge

Beginnend mit einer einelementigen Ausdrucksmenge werden alle Nachfolger dieses Ausdrucks bzgl. des ersten Reduktionstyps errechnet. Dazu werden alle möglichen Redexe des ersten Reduktionstyps markiert, und die markierten Terme reduziert. Das Ergebnis einer Reduktion kann bei internen Reduktionen eine Menge von Termen sein, so daß alle Simulationsergebnisse des Reduktionstyps `r` mittels `concat` zu einer einzigen Menge verschmolzen werden müssen. Diese Menge von reduzierten Termen wird jetzt rekursiv mit dem Rest der Reduktionstypen-Folge (`rs`) an `simulateReduction` übergeben. Es entsteht so eine Menge aller Ergebnisse aller möglichen Reduktionspfade einer Folge von Reduktionstypen, die wiederum konkateniert wird.

```
simulateReduction rec _ [] exprs
    = exprs

simulateReduction rec n (r:rs) exprs
    = concat [ simulateReduction rec (n + 1) rs es
             | es <- [descendants n r e | e <- exprs]
             ]
             where
               descendants n reds e
                 = concat [ reduceCoreExpr rec n e'
                          | e' <- markReduction rec r e
                          ]
```

Wegen der gleichzeitigen Simulation des Mengenkonstrukts müssen wir nun den aktuellen Reduktionstyp unterscheiden. Wenn eine Menge von Reduktionstypen (`SET as b c`) simuliert werden soll, dann erzeugt jeder Typ einen neuen Reduktionspfad bzw. -baum. Die Menge kann genau einmal ($\{\ldots\}$), keinmal oder mehrmals ($\{\ldots\}*$) oder mindestens einmal ($\{\ldots\}+$) wiederholt werden, so daß eine untere und eine obere Grenze der Wiederholungen angegeben werden muß, da sonst bei „unglücklich gewählten“ internen Reduktionstypen die Simulation unendlich lang werden kann[7].

[7] Der Konstrukt {`n.abs, i.cp, i.ldel`}+ bildet einen Baum von Reduktionspfaden, der bei entsprechenden Termen unendlich lange Reduktionspfade bzw. Reduktionen enthält.

```
simulateReduction rec _ [] exprs
    = exprs

simulateReduction rec n (r:rs) exprs
    = case r
        of (SET as b c) -> concat $
                           [ simulateReduction rec n rs exprs
                           | b == 0
                           ]
                           ++
                           [ simulateReduction rec (n + 1) rs' es
                           | es <- [descendants n as e | e <- exprs]
                           , b <= c
                           , let rs' = (SET as 0 (c - 1)):rs
                           ]
           otherwise    -> concat
                           [ simulateReduction rec (n + 1) rs es
                           | es <- [descendants n [r] e | e <- exprs]
                           ]
        where
           descendants n reds e
             = concat [ reduceCoreExpr rec n e'
                      | r' <- reds
                      , e' <- markReduction rec r' e
                      ]
```

Im Gegensatz zur ersten Version müssen hier die Nachfolger nicht eines, sondern mehrerer Reduktionstypen erzeugt werden, so daß der Funktion `descendants` eine Menge von Reduktionstypen übergeben werden muß.

Stößt `simulateReduction` auf einen Mengenkonstrukt, dann kann dieser möglicherweise die leere Menge sein (`b == 0`), so daß die Reduktionstypen der Menge nicht im Pfad enthalten sein müssen und die Simulation mit dem Rest der Folge fortgesetzt werden kann. Die zweite Menge (*list comprehension*) erzeugt die mehrfache Anwendung (bis zur oberen Grenze `c`) der im Mengenkonstrukt enthaltenen Typen[8].

6.4 Diagrammsuche

Wenn während der Validierung von Reduktionsdiagrammen Terme gefunden werden, für die kein passendes Diagramm im Diagrammsatz enthalten ist, müssen für diese Terme ein oder mehrere Diagramme ermittelt werden.

Die Funktion `deduceRules` ermittelt für einen Ausdruck Reduktionsdiagramme mit den entsprechenden internen Reduktionen auf der linken und rechten Seite (`iredsl` und `iredsr`). Auf beiden Seiten können natürlich auch alle NO-Reduktionen vorkommen.

[8] Hier sieht man auch, daß die (möglicherweise unnötige) Verwendung der *list comprehensions* der Lesbarkeit des Codes zugute kommt. Es würden sonst zu viele Unterscheidungen (`case` bzw. `if`) gemacht, die nur verwirren.

Das Prinzip zum Finden der Diagramme ist:

Definition 6.4.1 (Algorithmus *Deduce*) Sei t ein beliebiger Term.

1. $S := \{r = (i, a) \circ (no, r_1) \circ \cdots \circ (no, r_n) \mid n \geq 0, t \xrightarrow{r} t'\}$
   S ist also die Menge aller möglichen Reduktionsfolgen ausgehend von t, die mit der internen Reduktion (i, a) beginnen und in der Folge nur noch NO-Reduktionen enthalten.

2. $T := \{r = (no, r_1) \circ \cdots \circ (no, r_n) \circ (i, r_1) \circ \cdots \circ (i, r_m) \mid n, m \geq 0, t \xrightarrow{r} t'\}$
   Die Menge T ist die Menge aller Reduktionsfolgen ausgehend von t, die sowohl NO-Reduktionen als auch interne Reduktionen enthalten, aber die Folge der Reduktionen so geordnet ist, daß nach einer internen Reduktion keine NO-Reduktionen mehr folgen.

3. Erzeuge das Kreuzprodukt beider Mengen: $R = S \times T$.

4. Jedes Paar der erzeugten Menge R bildet ein Reduktionsdiagramm $(r_S \rightsquigarrow r_T)$, das mit dem Validierungs-Algorithmus *Validate* geprüft werden kann.

Der erste positiv validierte Diagrammsatz bildet den kleinstmöglichen, vollständigen Satz von Reduktionsdiagrammen, durch den der Term t geschlossen werden kann.

Die Bildung des Kreuzprodukts gestaltet sich deshalb etwas kompliziert, weil die Menge der Reduktionspfade unendlich sein kann, und deshalb das Kreuzprodukt mittels Diagonalisierung erzeugt werden muß. Die Funktion `getReductions` liefert nur bestimmte Folgen von Reduktionstypen, z. B. mit nur ausgewählten internen Reduktionstypen (`iredsl` bzw. `iredsr`). Weiterhin kann bestimmt werden, ob NO-Reduktionen nach internen Reduktionen folgen dürfen und umgekehrt. Dadurch werden Diagramme ausgeschlossen, die nicht in das gesuchte Raster passen und dadurch umsonst geprüft würden. Der Parameter `allrules` bestimmt, ob diese Filterung ein- oder ausgeschaltet ist, `searchMax` begrenzt die Diagrammsuche.

```
deduceRules rec searchMax (iredsl,iredsr) allrules expr
    = case closeExpression rec startReds genRules expr
        of RuleNotFound  -> []
           RulesFound rs -> rs
        where
          startReds
            = [I r | r <- iredsl]

          genRules
            = [ (i:lhs,rhs)
              | i <- startReds
              , let lexprs = concat [ reduceCoreExpr rec 0 e
                                    | e <- markReduction rec i expr
                                    ]
              , (l,r) <- take searchMax $ rulesf lexprs [expr]
              , let (lhs,rhs) = (map makeGeneralReduction l
                                ,map makeGeneralReduction r)
              , (i:lhs) /= rhs
              ]
```

```
rulesf = if allrules then allRules else comRules

comRules es1 es2
  = diag (getReductions rec []     False False es1) -- n*
         (getReductions rec iredsr False True  es2) -- n* i*

allRules es1 es2
  = diag (getReductions rec iredsl True True es1)
         (getReductions rec iredsr True True es2)
```

Erzeugen möglicher Reduktionsfolgen

Die in `deduceRules` benutzte Funktion `getReductions` erzeugt, von einem Term ausgehend, die Menge aller möglichen Reduktionsfolgen, die NO-Reduktionen und bestimmte interne Reduktionen enthalten. Da wir bei der Diagrammsuche triviale (z. B. $(i,a) \circ (no,b) \rightsquigarrow (i,a) \circ (no,b)$) und uninteressante Fälle (z. B. $(i,a) \circ (i,b) \rightsquigarrow (i,c)$) ausschließen wollen, werden durch die Parameter `i_no` und `no_i` die Reduktionsfolgen eingeschränkt. Dabei gibt `i_no` an, ob nach einer internen Reduktion noch NO-Reduktionen folgen dürfen, und `no_i` erlaubt interne Reduktionen nach einer NO-Reduktion. Die zu verwendenden internen Reduktionen sind in der Liste `ireds` spezifiziert. So können beide Seiten des Diagramms genau charakterisiert werden.

```
getReductions rec ireds i_no no_i exprs
    = []:[ path | (path,_) <- sim 1 [([],exprs)]]
        where
          sim n xs = case concat [ step n rs e
                                 | (rs, es) <- xs
                                 , e        <- es
                                 ]
                       of []    -> []
                          paths -> paths ++ (sim (n + 1) paths)
```

Alle Pfade werden um einen Schritt erweitert, wobei vorher geprüft werden muß, ob eine NO-Reduktion möglich ist, so daß der NO-Redex aus den internen Redexen entfernt werden kann.

```
step n rs e
  = [(rs ++ [r], es) | (r,es) <- reds]
    where
      reds = case markNORedex rec e
               of HasRedex r e' -> (redsNO r e') ++ (redsI (/= e'))
                  otherwise     -> redsI (\x -> True)
```

Bei einer (nd)-Reduktion müssen wir die beiden Varianten (ndl) und (ndr) getrennt erzeugen.

```
redsNO ND e' = [ ((NO r'), reduceCoreExpr rec n e'')
               | i_no || all isNOreduction rs
               , r' <- [NDL,NDR]
               , let e'' = replaceND r' e'
               ]
```

Der in diesem Fall eindeutige NO-Redex wird nur in den Pfad aufgenommen, wenn NO-Reduktionen erlaubt sind oder bisher nur NO-Reduktionen im Pfad sind.

```
redsNO r  e' = [ ((NO r), reduceCoreExpr rec n e')
               | i_no || all isNOreduction rs
               ]
```

Interne Reduktionen dürfen nur in den Pfad übernommen werden, wenn es erlaubt ist (`no_i`) oder bisher nur interne Reduktionen im Pfad sind. Die Funktion `f` filtert bestimmte markierte Ausdrücke heraus, z. B. den NO-Redex.

```
redsI f = [ (I t, eIs)
          | no_i || all isIreduction rs
          , t <- ireds
          , let eIs = concat [ reduceCoreExpr rec n eI
                             | eI <- markRedex rec t e
                             , f eI
                             ]
          , eIs /= []
          ]
```

Die Reduktionspfade werden durch die Unterfunktion `sim` folgendermaßen erzeugt:

1. Alle Ausdrücke in `exprs` (am Anfang ist die Menge einelementig) werden in einem Schritt mittels aller möglichen und erlaubten Reduktionen (Normalordnung und intern) reduziert.

2. Der Reduktionstyp wird mit dem resultierenden Ausdruck gespeichert.

3. Alle Ausdrücke werden mit ihren zugewiesenen Pfaden rekursiv an `sim` übergeben.

Das Ergebnis ist eine (potentiell unendliche) Liste von Reduktionspfaden, geordnet nach der Länge der Reduktionsfolge.

Kapitel 7

Ergebnisse

Wir geben für gängige Programmtransformationen Reduktionsdiagramme an, die wir jedoch als erweiterte Reduktionsregeln der Basis-Kalküle λ_{nd} und $\lambda_{nd,rec}$ implementieren. Diese Reduktionsdiagramme sollen dann mit JONAH validiert werden. Wir werden sehen, daß es bei einigen Diagrammsätzen — speziell der (cp)-Reduktion — Schwierigkeiten gibt, so daß weitere Programmtransformationen (z. B. $(ldup)$) hinzugefügt und validiert werden müssen.

Als Folge der Schwierigkeiten mit der (cp)- und $(ldup)$-Reduktion soll der Basis-Kalkül $\lambda_{nd,rec}$ derart ergänzt werden — hier handelt es sich um eine echte Erweiterung der Basis-Kalkül-Regeln —, so daß diese Schwierigkeiten umgangen werden können. Es werden zwei Varianten des Kalkül $\lambda_{nd,rec}$ beschrieben, die ohne $(ldup)$-Reduktion auskommen sollen.

7.1 Validierung der Reduktionsdiagramme

Wie in Kapitel 5.3.3 zu sehen ist, sind die Mengen der zu testenden Ausdrücke sehr groß. Aus diesem Grund ist die Validierung der Reduktionsdiagramme sehr zeitaufwendig. Wir können zwar mehrere Diagrammsätze gleichzeitig validieren, doch auch dann wird nach einer langen Rechenzeit „nur bestätigt“, daß alle Terme durch ein Diagramm des jeweiligen Satzes geschlossen werden können.

In der Praxis hat sich JONAH allerdings darin bewährt, daß schon bei kleinen Termmengen fehlende Diagramme schnell angemahnt werden. Die entsprechenden Terme können dann als Grundlage für die Entwicklung weiterer Diagramme dienen. In einigen Fällen können sogar vorhandene Diagramme derart verallgemeinert werden, so daß sich an der Anzahl der Reduktionsdiagramme nichts geändert hat.

Die (cp)-Reduktion ist, wie schon oft angedeutet, der schwierigste Fall. Trotz Hinzunahme der $(ldup)$-, $(ldel)$ oder (eql)-Transformationen gibt es immer wieder Terme, die nicht geschlossen werden können.

Wir verwenden bei allen Tests nur den Datentyp `List`, da er den nullstelligen Konstruktor `Nil` und den zweistelligen Konstruktor `Cons` enthält. Dies reicht für die meisten Untersuchungen aus, besonders für die (cp)-basierten Transformationen.

Wegen der oben genannten Gründe werden wir nur an wenigen Stellen die Ergebnisse der Validierung beschreiben. Dabei werden wir die Termklasse $(1, 2, 2, 1, 0)$ betrachten, da diese verhältnismäßig viele verschiedene Termstrukturen, andererseits jedoch nicht so viele Terme enthält. Die Terme mit `choice`-Ausdrücken überprüfen wir mit der Termklasse $(1, 1, 2, 1, 1)$, die zwar auch sehr viele Terme abdeckt, jedoch in annehmbarer Zeit validiert werden kann.

```
$ jonah -H80M validate --ruleset=llet,ldel,lcom,lcv --types=List
  --time --verbose --stats=10000 --addrules --class=1,1,2,1,1

--rulesets=llet,ldel,lcom,lcv
-- 20,007,233 expressions in class (1, 1, 2, 1, 1).
-- 20,007,233 expressions tested.
-- rule set   |      tested expressions |  not found
----------------------------------------------------
-- llet       |    401,448 |      2.0 % |          0
-- ldel       |  3,038,013 |     15.1 % |          0
-- lcom       |  1,401,088 |      7.0 % |          0
-- lcv        |    340,682 |      1.7 % |          0
Execution time: 4 h, 28 m, 31 s
```

In den folgenden Kapiteln werden Transformationen und Reduktionsdiagramme gemäß der zeitlichen Reihenfolge ihrer Entwicklung aufgeführt. Das heißt, daß wir mit dem nicht-rekursiven Kalkül beginnen, mit dem rekursiven Kalkül fortfahren und aus den Erfahrungen und Mißerfolgen, die neuen Transformationen entwickeln. Die Entwicklung mündet in einen Kalkül mit neu definierten (cp)- und $(case)$-Reduktionsregeln. Die Angabe vollständiger Diagrammsätze müssen wir jedoch schuldig bleiben, da die Zeit für die umfangreichen Validierungen nicht mehr ausreichte.

7.2 Reduktionsdiagramme im λ_{nd}-Kalkül

Auf Seite 59 haben wir die kontextuelle Äquivalenz der Reduktionsregeln $(lbeta)$, (abs), $(lapp)$, $(lcase)$ und $(case)$ gezeigt. Fehlen jetzt also nur noch die Regeln (ndl/r), $(llet)$ und (cp). Erstere können per Definition die kontextuelle Äquivalenz nicht erhalten und die kontextuelle Äquivalenz für die $(llet)$-Reduktion kann nach Angabe von Sätzen von Vertauschungs- und Gabeldiagrammen gezeigt werden. Die (cp)-Reduktion macht jedoch, wie oben schon angedeutet, Schwierigkeiten. Wir werden nur mit Hilfe von zusätzlichen Programmtransformationen (neuen Reduktionsregeln) Vertauschungs- und Gabeldiagramme konstruieren können, so daß die kontextuelle Äquivalenz der (cp)-Reduktion gezeigt werden kann.

7.2.1 Reduktionsdiagramme für $(llet)$

Lemma 7.2.1 Ein kompletter Satz Vertauschungsdiagramme für die $(llet)$-Reduktion enthält folgende Diagramme:

- $(i, llet) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, llet)$

- $(i, llet) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (no, llet)$
- $(i, llet) \circ (no, \{llet, lapp, lcase\}^+) \quad \rightsquigarrow \quad (no, \{llet, lapp, lcase\}^+) \circ (i, llet)^{0 \vee 1}$

Lemma 7.2.2 Ein kompletter Satz Gabeldiagramme für die (llet)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,llet} \quad \rightsquigarrow \quad \xrightarrow{i,llet} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,llet} \circ \xleftarrow{no,a} \circ \xrightarrow{i,llet} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
- $\xleftarrow{no,\{llet,lapp,lcase\}^+} \circ \xrightarrow{i,llet} \quad \rightsquigarrow \quad \xrightarrow{i,llet} \circ \xleftarrow{no,\{llet,lapp,lcase\}^+}$
- $\xleftarrow{no,\{llet,lapp,lcase\}^+} \circ \xrightarrow{i,llet} \quad \rightsquigarrow \quad \xleftarrow{no,\{llet,lapp,lcase\}^+}$

Validierung der ($llet$)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert bei kleinen Termklassen keine Auffälligkeiten.

```
$ jonah -H80M validate --ruleset=llet --types=List
   --addrules --time --verbose --stats=10000 --class=1,2,2,1
```

erzeugt die Ausgabe[1,2]:

```
--rulesets=llet
-- 9,677,610 expressions in class (1, 2, 2, 1, 0).
-- 9,677,610 expressions tested.
-- rule set   |      tested expressions |  not found
------------------------------------------------------
-- llet       |    177,727 |      1.8 % |          0
Execution time: 1 h, 32 m, 15 s
```

Wir wollen nun einmal versuchsweise das dritte Vertauschungsdiagramm aus dem Diagrammsatz entfernen, bzw. in der Regeldatei auskommentieren:

```
-- > i.llet {n.llet, n.lapp, n.lcase}+ ==> {n.llet, n.lapp, n.lcase}+ (i.llet)?
```

[1]Die Ausführungszeiten sind in den folgenden Beispielen gleich, da wir vier Diagrammsätze gleichzeitig überprüft haben und die Ergebnisse hier für jede Transformation aufschlüsseln.

[2]Die genaue Befehlssyntax von Jonah findet sich in Anhang C.

```
$ jonah -H80M validate --ruleset=llet --types=List --addrules --time --verbose
  --stats=10000 --class=1,2,2

--rulesets=llet
-- 37,923 expressions in class (1, 2, 2, 0, 0).
-- llet: ((let a=(let b=Nil in b) in a) Nil)
> i.llet n.lapp n.lapp ==> n.lapp n.llet

-- llet: ((let a=(let b=Nil in b) in Nil) Nil)
> i.llet n.lapp n.lapp ==> n.lapp i.llet


-- llet: (((let a=(let b=Nil in b) in a) Nil) Nil)
> i.llet n.lapp n.lapp n.lapp n.lapp ==> n.lapp n.lapp n.llet

-- llet: (((let a=(let b=Nil in b) in Nil) Nil) Nil)
> i.llet n.lapp n.lapp n.lapp n.lapp ==> n.lapp n.lapp i.llet


-- 37,923 expressions tested.
-- rule set   |      tested expressions |  not found
---------------------------------------------------
-- llet       |      1,305 |      3.4 % |          4
Execution time: 7 s
```

Es werden für vier Terme keine passenden Reduktionsdiagramme im Diagrammsatz gefunden. Jonah gibt diese Terme aus und ermittelt für jeden Term ein minimales Vertauschungsdiagramm. Obwohl die Terme nicht wohlgeformt sein müssen, so kann doch an der Struktur eines Terms erkannt werden, welches Diagramm wirklich benötigt wird. D. h. durch Austauschen der Konstruktoren kann ein wohlgeformter Term erzeugt werden:

(`let` a = (`let` b = `Nil` `in` b) `in` a) `Nil` ist nicht wohlgeformt,

(`let` a = (`let` b = `Succ` `in` b) `in` a) `Zero` dagegen schon.

Anhand der Struktur der ermittelten Diagramme kann nun geprüft werden, für welche Terme solche Diagramme benötigt werden. Die verallgemeinerte Form dieser Diagramme stellt das auskommentierte Diagramm dar.

Mit der Annahme, daß die oben definierten Diagrammsätze vollständig sind, kann die folgende Behauptung bewiesen werden.

Behauptung 7.2.1 Wenn $s \xrightarrow{i,llet} t$, dann $s \sim_C t$.

Beweis. Wir nehmen an, daß die interne $(llet)$-Reduktion sich in einem Reduktionskontext befindet. D. h. $s \equiv R[s']$ und s' ist ein $(llet)$-Redex.

1. Existenz einer WHNF:
   Habe t eine NO-Reduktion zu einer WHNF. Induktion über die Länge der NO-Reduktion zeigt, daß s ebenfalls eine NO-Reduktion zu einer WHNF besitzt, da eine $(llet)$-Reduktion

eine WHNF nicht zerstören kann. Habe t eine NO-Reduktion zu einer WHNF. Dann kann ebenfalls per Induktion über die Länge der NO-Reduktion gezeigt werden, daß t eine NO-Reduktion zu einer WHNF besitzt. Der Beweis durch Induktion kann nur funktionieren, wenn nach jedem Reduktionsschritt ein Vertauschungs- oder Gabeldiagramm angewendet werden kann.

2. Alle NO-Reduktionen terminieren:
   Wir müssen hier wie oben für alle NO-Reduktionen das Terminieren beweisen. Dazu wird als Induktionsparameter eine Multimenge der NO-Reduktionslängen benutzt.

Insgesamt gilt also $s \sim_C t$. Wegen der Kongruenz-Eigenschaft von $\sim_C$ ergibt sich, daß die (*llet*)-Reduktion überall im Term angewandt werden darf. □

7.2.2 Reduktionsdiagramme für (*ldel*)

Während der Reduktion eines Terms werden immer wieder `let`-Bindungen erzeugt, die im weiteren Verlauf nicht mehr referenziert werden. Nicht nur in funktionalen Programmiersprachen wird daher während der Ausführung eines Programms von Zeit zu Zeit nach solchen „speicherfressenden“ Daten gesucht und diese dann gelöscht. Dieser Vorgang wird gemeinhin als *garbage collection*[3] bezeichnet.

Wir wollen nun zeigen, daß diese Programmtransformation korrekt ist und erweitern dazu den Basis-Kalkül durch die Reduktionsregel (*ldel*), die eine `let`-Bindung entfernt, wenn die Variable der Bindung im Ausdruck nicht mehr referenziert wird.

Definition 7.2.1 Sei C ein beliebiger Kontext. Die (*ldel*)-Reduktion ist definiert durch:
$C[\ \texttt{let}\ x\ \texttt{=}\ s\ \texttt{in}\ t\ ] \xrightarrow{ldel} C[\ t\ ]$, falls x nicht in t vorkommt.

Markierung von (*ldel*)

Die Markierung eines (*ldel*)-Redex gestaltet sich sehr einfach, da nur überprüft werden muß, ob die `let`-gebundene Variable im Term `expr` frei vorkommt. Ist dies nicht der Fall, dann kann der `let`-Ausdruck als (*ldel*)-Redex markiert werden.

```
markLDel (ELet _ rec@False bs expr)
    = [ ELet (Redex (LDel var)) rec bs expr
      | var <- [v  | (v,_) <- bs]
      , var `notElem` getFreeVars expr
      ]
markLDel expr = []
```

Es scheint hier etwas umständlich, daß die Variable `var` aus einer Menge von Bindungen deklariert wird. Wir wollen jedoch diese Funktion später noch für `letrec`-Ausdrücke verwenden, die mehrere Bindungen enthalten können.

[3]engl.: „Abfall sammeln“

Reduktion von (*ldel*)

Die Reduktion eines (*ldel*)-Redex ist ebenfalls trivial. Hier verwenden wir *pattern matching*, um nur den nicht-rekursiven Fall (`False`) zu behandeln, der genau eine Bindung enthält (`[(bv,be)]`).

```
reduceCoreExpr False level (ELet (Redex (LDel v)) r [(bv,be)] e)
    = [e]
```

Diagrammsätze

Wir geben je einen kompletten Satz Vertauschungs- und Gabeldiagramme an, die für den Beweis der kontextuellen Äquivalenz der (*ldel*)-Reduktion benötigt werden.

Lemma 7.2.3 Ein kompletter Satz Vertauschungsdiagramme für die (*ldel*)-Reduktion enthält folgende Diagramme:

- $(i, ldel) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, ldel)$
- $(i, ldel) \quad \rightsquigarrow \quad (no, \{lapp, llet, lcase\}^+) \circ (i, ldel)$
- $(i, ldel) \circ (no, \{lapp, llet, lcase\}^+) \quad \rightsquigarrow \quad (no, \{lapp, llet, lcase\}^*) \circ (i, ldel)$

Lemma 7.2.4 Ein kompletter Satz Gabeldiagramme für die (*ldel*)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,ldel} \quad \rightsquigarrow \quad \xrightarrow{i,ldel} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,\{lapp,llet,lcase\}^+} \circ \xrightarrow{i,ldel} \quad \rightsquigarrow \quad \xrightarrow{i,ldel}$
- $\xleftarrow{no,\{lapp,llet,lcase\}^+} \circ \xrightarrow{i,ldel} \quad \rightsquigarrow \quad \xrightarrow{i,ldel} \circ \xleftarrow{no,\{lapp,llet,lcase\}^*}$

Validierung der (*ldel*)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert bei kleinen Termklassen nur geringfügige Auffälligkeiten.

```
$ jonah -H80M validate --ruleset=ldel --types=List
  --time --verbose --stats=10000 --addrules --class=1,2,2,1
```

erzeugt die Ausgabe:

```
--rulesets=ldel
-- 9,677,610 expressions in class (1, 2, 2, 1, 0).
-- 9,677,610 expressions tested.
-- rule set   |      tested expressions |  not found
----------------------------------------------------
-- ldel       |  1,590,673 |     16.4 % |          0
Execution time: 1 h, 32 m, 15 s
```

7.2.3 Reduktionsdiagramme für (*lcv*)

Die Programmtransformation (*lcv*), entfernt Referenzen auf Variablen, indem diese Referenz auf die Variable direkt im Ausdruck durch die Variable selbst ersetzt werden. Im Prinzip ist eine (*lcv*)-Reduktion eine (*cp*)-Reduktion, die nur Variablen kopiert.

Die (*lcv*)-Reduktion ist in die zwei Teilreduktionen (*lcvt*) und (*lcvd*) unterteilt, die sich darin unterscheiden, daß (*lcvd*) auch in Abstraktionen kopieren kann.

Definition 7.2.2 Seien C, D, D' beliebige Kontexte und S ein Oberflächenkontext. Die (*lcvt*)- und (*lcvd*)-Reduktion[4] sind definiert durch:

$$C[\ \texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ S[\ x\ ]\ ] \xrightarrow{lcvt} C[\ \texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ S[\ y\ ]\ ]$$

$$C[\ \texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ D[\ \lambda z\,.\,D'[x]\ ]\ ] \xrightarrow{lcvd} C[\ \texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ D[\ \lambda z\,.\,D'[y]\ ]\ ]$$

Markierung von (*lcvt*) und (*lcvd*)

Wir sammeln alle `let`-gebundenen Variablen, deren gebundener Ausdruck eine Variable ist, und markieren diese im Ausdruck `expr`. Alle möglichen Redex werden dadurch erzeugt. Ein `let`-Ausdruck hat zwar nur eine Bindung, die beiden Funktionen können aber mit mehreren Bindungen umgehen, da wir sie im rekursiven Kalkül in Kapitel 7.3.2 entsprechend erweitern.

```
markLCVT (ELet _ rec@False bs expr)
    = [ ELet (Redex (LCVT var)) rec bs expr'
      | var   <- cpvars
      , expr' <- markContextS (markVar var) expr
      ]
      where
        cpvars = [ v
                 | (v,e) <- bs
                 , isVariable e
                 , v /= getVariableName e
                 ]
markLCVT expr = []
```

Ein (*lcvd*)-Redex kann sich nur unter einer Abstraktion befinden. Deshalb markieren wir die Zielvariable erst in allen Kontexten und entfernen aus dieser Termmenge die in Oberflächenkontexten markierten Zielvariablen.

[4] let copy variable

```
markLCVD (ELet _ rec@False bs expr)
    = [ ELet (Redex (LCVD var)) rec bs expr'
      | var   <- cpvars
      , expr' <- markContextC (markVar var) (True,True) expr
                 \\
                 markContextsS (markVar var) expr
      ]
      where
        cpvars = [ v
                 | (v,e) <- bs
                 , isVariable e
                 , v /= getVariableName e
                 ]
markLCVD expr = []
```

Reduktion von (*lcvt*) und (*lcvd*)

Zur Reduktion eines (*lcvt*)- bzw. (*lcvd*)-Redex ändern wir den Redex in einen (*cp*)-Redex und reduzieren diesen in entsprechender Weise.

```
reduceCoreExpr rec level (ELet (Redex (LCVT v)) r bs e)
    = reduceCoreExpr rec level (ELet (Redex (Copy v)) r bs e)

reduceCoreExpr rec level (ELet (Redex (LCVD v)) r bs e)
    = reduceCoreExpr rec level (ELet (Redex (Copy v)) r bs e)
```

Diagrammsätze

Lemma 7.2.5 Ein kompletter Satz Vertauschungsdiagramme für die (*lcv*)-Reduktion enthält folgende Diagramme:

- $(i, lcvt) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, lcv)$
- $(i, lcvt) \circ (no, cp) \quad \rightsquigarrow \quad (no, cp) \circ (no, cp) \circ \xleftarrow{i,cpt}$
- $(i, lcvt) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, lcvd) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, lcvd)$
- $(i, lcvd) \circ (no, cp) \quad \rightsquigarrow \quad (no, cp) \circ (i, lcvd) \circ (i, lcvd)$
- $(i, lcvd) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, lcvd) \circ (no, lbeta) \quad \rightsquigarrow \quad (no, lbeta) \circ (i, lcvt)$

Lemma 7.2.6 Ein kompletter Satz Gabeldiagramme für die (*lcom*)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,lcv} \;\leadsto\; \xrightarrow{i,lcv} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cp} \circ \xrightarrow{i,lcv} \;\leadsto\; \xrightarrow{i,lcv} \circ \xrightarrow{i,lcv} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cp} \circ \xleftarrow{no,cp} \circ \xrightarrow{i,lcv} \;\leadsto\; \xrightarrow{i,lcv} \circ \xleftarrow{no,cp}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,lcvd} \;\leadsto\; \xrightarrow{i,lcvd} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,lcvd} \;\leadsto\; \xrightarrow{i,lcv} \circ \xleftarrow{no,a}$
  mit $a \in \{lbeta, case\}$

Validierung der (*lcv*)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert bei kleinen Termklassen keine Auffälligkeiten.

```
$ jonah -H80M validate --ruleset=lcv --types=List
  --time --verbose --stats=10000 --addrules --class=1,2,2,1
```

erzeugt die Ausgabe:

```
--rulesets=lcv
-- 9,677,610 expressions in class (1, 2, 2, 1, 0).
-- 9,677,610 expressions tested.
-- rule set   |      tested expressions |  not found
---------------------------------------------------
-- lcv        |    202,549 |      2.0 % |          0
Execution time: 1 h, 32 m, 15 s
```

7.2.4 Reduktionsdiagramme für (*lcom*)

Zur Vertauschung von `let`-Ausdrücken definieren wir die Reduktion (*lcom*)[5]. Diese Programmtransformation hat nur im nicht-rekursiven Kalkül einen Sinn, da Bindungen in `letrec`-Ausdrücken per Definition vertauschbar sind.

Definition 7.2.3 Sei C ein beliebiger Kontext. Die (*lcom*)-Reduktion ist definiert durch:

$$C[\;\texttt{let}\; x = t_x \;\texttt{in let}\; y = t_y \;\texttt{in}\; s\;] \xrightarrow{lcom} C[\;\texttt{let}\; y = t_y \;\texttt{in let}\; x = t_x \;\texttt{in}\; s\;],$$

falls x nicht frei in t_y vorkommt.

Markierung von (*lcom*)

Zur Markierung kommen nur zwei hintereinander geschaltete `let`-Ausdrücke infrage. Zu prüfen ist, ob die `let`-gebundene Variable des äußeren `let`-Ausdrucks im gebundenen Term

[5] let commutation

des inneren `let`-Ausdrucks frei vorkommt. Falls nicht, dann können die `let`-Ausdrücke vertauscht werden. Allen anderen Ausdrücke — auch `letrec`-Ausdrücke (siehe `False`) — werden ignoriert und bleiben unmarkiert.

```
markLCom (ELet _ False [(x,xe)] (ELet _ False [(y,ye)] e))
    = if x `notElem` (getFreeVars ye)
         then [ELet (Redex LCom) False [(x,xe)] (ELet Unmarked False [(y,ye)] e)]
         else []
markLCom expr = []
```

Reduktion von (*lcom*)

Wir müssen nur den nicht-rekursiven Fall betrachten. Sicherheitshalber wird noch der rekursive Fall aufgeführt, der jedoch keine Redukte produziert.

```
reduceCoreExpr False level (ELet (Redex LCom) r bs e@(ELet _ _ bs' e'))
    = [ELet Unmarked False bs' (ELet Unmarked False bs e')]

reduceCoreExpr True level (ELet (Redex LCom) r bs e)
    = []
```

Diagrammsätze

Lemma 7.2.7 Ein kompletter Satz Vertauschungsdiagramme für die (*lcom*)-Reduktion enthält folgende Diagramme:

- $(i, lcom) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, lcom)$
- $(i, lcom) \circ (no, \{llet, lapp, lcase\}^{+}) \quad \rightsquigarrow \quad (no, \{llet, lapp, lcase\}^{*}) \circ (i, lcom)$

Lemma 7.2.8 Ein kompletter Satz Gabeldiagramme für die (*lcom*)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,lcom} \quad \rightsquigarrow \quad \xrightarrow{i,lcom} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,\{llet,lapp,lcase\}^*} \circ \xrightarrow{i,lcom} \quad \rightsquigarrow \quad \xrightarrow{i,lcom} \circ \xleftarrow{no,\{llet,lapp,lcase\}^*}$

Validierung der (*lcom*)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert bei kleinen Termklassen keine Auffälligkeiten.

```
$ jonah -H80M validate --ruleset=lcom --types=List
  --time --verbose --stats=10000 --addrules --class=1,2,2,1
```

erzeugt die Ausgabe:

```
--rulesets=lcom
-- 9,677,610 expressions in class (1, 2, 2, 1, 0).
-- 9,677,610 expressions tested.
-- rule set   |      tested expressions |  not found
-----------------------------------------------------
-- lcom       |    718,528 |      7.4 % |          0
Execution time: 1 h, 32 m, 15 s
```

7.2.5 Reduktionsdiagramme für (*ldup*)

Durch das Kopieren von Termen in Konstruktoranwendungen und das daran anschließende Abstrahieren dieser Kopie, kann es dazu kommen, daß verschiedene `let`-gebundene Variablen denselben Term (bzgl. α-Konvertibilität) an sich binden. Das Kopieren und darauf folgende Abstrahieren soll durch die neue Reduktionsregel (*ldup*)[6] kompensiert werden.

Die Definition von (*ldup*) ist sehr allgemein gehalten und die Anzahl der resultierenden Redukte ist sehr groß.

Definition 7.2.4 Sei C ein beliebiger Kontext und S, S' Oberflächenkontexte. Die *(ldup)-Reduktion* ist definiert durch:

$$C[\ \texttt{let}\ x = t\ \texttt{in}\ S[\ S'[x, ..., x]\ ]\ ] \xrightarrow{ldup} C[\ \texttt{let}\ x = t\ \texttt{in}\ S[\ \texttt{let}\ y = t\ \texttt{in}\ S'[z_1, ..., z_n]\ ]\ ],$$

falls t kopierbar, $z_i \in \{x, y\}$ und y neu.

Es wird in allen Oberflächenkontexten alle möglichen Variablenbelegungen (x oder y) erzeugt.

Markierung von (*ldup*)

Markiert werden alle `let`-Ausdrücke deren `let`-gebundene Variable kopierbar ist. Zu beachten ist, daß sich die Variable nicht unbedingt im Ausdruck `expr` befinden muß.

```
markLDup (ELet _ rec bs expr)
    = let cpvars = [v | (v,e) <- bs, isPureConstructor e || isLambda e]
       in [ ELet (Redex (LDup var)) rec bs expr
          | var <- cpvars
          ]
markLDup expr = []
```

Reduktion von (*ldup*)

Die (*ldup*)-Reduktion ist daher sehr kompliziert, da das Verdoppeln der Variablen, und damit das Einsetzen eines `let`-Terms, an vielen Stellen des Terms `e` und innerhalb der Bindungen (bei `letrec`) realisiert werden muß. Dazu dient die rekursive Funktion `setv'`, die alle möglichen Belegungen der Variablen `v` und `v'` in dem Zielterm erzeugt. Wir verwenden dazu die

[6] *let duplication*

Funktionen `markContextS`, `applyFunc` und `renameVar`, die normalerweise nur zum Markieren eines Terms benutzt werden.

```
reduceCoreExpr rec level (ELet (Redex (LDup v)) r bs e)
    = [ ELet Unmarked False bs e'
      | e' <- markContextS insertLet e
      ]
      where
```

Zuerst wird die neue Variable `v'` und die neue Bindung erzeugt. An jeder Stelle des Terms `e` muß der neue `let`-Ausdruck plaziert werden. Dazu bedienen wir uns der Markierungsfunktion für Oberflächenkontexte (`markContextS`) und der Pseudo-Markierung `insertLet`, die alle Möglichkeiten der Variablenbelegung innerhalb eines Kontextes erzeugt.

```
be          = lookupBinding v bs

v'          = newVarPrefix ++ show level

bs'         = [(v',be)]

insertLet x =    [ELet Unmarked rec bs' x]
              ++ [ELet Unmarked rec bs' y | y <- nub (setv' [x])]
```

Für die Variablenbelegungen ist die Funktion `setv'` zuständig. Sie benennt sukzessive alle Vorkommen der Variable `v` in `v'` um. Dabei werden die bisher umbenannten Terme rekursiv an `setv'` übergeben, um alle Permutationen der Belegungen zu erhalten.

```
setv' []     = []
setv' (x:xs) = let xs' = markContextS (markVar v) x
                   ys  = [applyFunc renameVar x' | x' <- xs']
               in ys ++ setv' (ys ++ xs)

renameVar e@(EVar VWHNF var)
  | v == var  = (EVar Unmarked v')
  | otherwise = e
renameVar e   = e
```

Diagrammsätze

Lemma 7.2.9 Ein kompletter Satz Vertauschungsdiagramme für die ($ldup$)-Reduktion enthält folgende Diagramme:

- $(i, ldup) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, ldup)$
- $(i, ldup) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, ldup) \circ (no, \{llet, lapp, lcase\}^{+}) \quad \rightsquigarrow \quad (no, \{llet, lapp, lcase\}^{*}) \circ (i, ldup)$

Lemma 7.2.10 Ein kompletter Satz Gabeldiagramme für die ($ldup$)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,ldup} \quad \rightsquigarrow \quad \xrightarrow{i,ldup} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,ldup} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,\{llet,lapp,lcase\}^*} \circ \xrightarrow{i,ldup} \quad \rightsquigarrow \quad \xrightarrow{i,ldup} \circ \xleftarrow{no,\{llet,lapp,lcase\}^*}$

Validierung der (*ldup*)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert bei kleinen Termklassen keine Auffälligkeiten.

```
$ jonah -RTS validate --ruleset=ldup --redexes=all
--types=List --time --verbose --addrules --class=1,1,2
```

erzeugt die Ausgabe:

```
--rulesets=ldup
-- 37,923 expressions in class (1, 2, 2, 0, 0).
-- ldup: no rule for ‘(let a=Nil in (let b=(Cons a) in b))‘
-- ERROR: ≪ldup/a:(let a=Nil in (let b=(Cons a) in b))≫
      --> (let a=Nil in (let b=(Cons (let new_1=Nil in a)) in b))
        : No rule found!
-- ERROR: ≪ldup/a:(let a=Nil in (let b=(Cons a) in b))≫
      --> (let a=Nil in (let b=(Cons (let new_1=Nil in new_1)) in b))
        : No rule found!

[...]

-- 37,923 expressions tested.
-- rule set   |      tested expressions |  not found
---------------------------------------------------
-- ldup       |     22,547 |     59.4 % |       299
Execution time: 5 m, 50 s
```

299 der 22.547 getesteten Terme — wir haben nur den ersten aufgeführt — können nicht durch die Diagramme des Diagrammsatzes geschlossen werden. Für die letzten beiden Redukte der acht möglichen findet sich kein Diagramm im Satz, und Jonah kann trotz zusätzlicher Verwendung aller Transformationen (`--redexes=all`) kein passendes Diagramm erzeugen.

Daraus ergibt sich, daß wahrscheinlich sehr komplizierte Reduktionsdiagramme für die (*ldup*)-Reduktion existieren, wohl möglich durch Hinzunahme weiterer Transformationen. Es könnte daran liegen, daß die (*ldup*)-Reduktion wie eine (*cp*)-Reduktion wirkt, und somit für die (*ldup*)- und(*cp*)-Diagramme eine gegenseitige Abhängigkeit entsteht.

Wir stellen im folgenden Kapitel dennoch die (nicht-vollständigen) Reduktionsdiagramme der (*cp*)-Reduktion vor.

7.2.6 Reduktionsdiagramme für (cp)

Die (cp)-Reduktion ist aufgeteilt in zwei Unterreduktionen, die sich darin unterscheiden, daß die eine (cpd) unterhalb von Abstraktionen kopiert, die andere (cpt) jedoch nicht.

Definition 7.2.5 Seien C, D, D' beliebige Kontexte, c ein Konstruktor und t ein kopierbarer Term (Abstraktion oder Konstruktoranwendung). Die (cpd)- und (cpt)-Reduktion sind definiert durch:

$$C[\ \texttt{let}\ x = t\ \texttt{in}\ D[\ \lambda y\,.\,D'[x]\ ]\ ] \xrightarrow{cpd} C[\ \texttt{let}\ x = t\ \texttt{in}\ D[\ \lambda y\,.\,D'[t]\ ]\ ]$$

$$C[\ \texttt{let}\ x = t\ \texttt{in}\ S[x]\ ] \xrightarrow{cpt} C[\ \texttt{let}\ x = t\ \texttt{in}\ S[t]\ ]$$

Die Markierung der beiden Redex-Typen ist ausführlich in Kapitel 3.3.3 auf Seite 47 beschrieben.

Die (cpt)- und (cpd)-Reduktionen können wie ein normales (cp) reduziert werden. Wir müssen sie nur bei der Validierung unterscheiden. Zur Reduktion ändern wir einfach nur die (cpt)- bzw. (cpd)-Markierung zu (cp) und reduzieren als (cp).

```
reduceCoreExpr rec level (ELet (Redex (CopyD v)) r bs e)
    = reduceCoreExpr rec level (ELet (Redex (Copy v)) r bs e)

reduceCoreExpr rec level (ELet (Redex (CopyT v)) r bs e)
    = reduceCoreExpr rec level (ELet (Redex (Copy v)) r bs e)
```

Diagrammsätze

Wir geben je einen vollständigen Satz Vertauschungs- und Gabeldiagramme an:

Lemma 7.2.11 Ein kompletter Satz Vertauschungsdiagramme für die (cpd)- und (cpt)-Reduktion enthält folgende Diagramme:

- $(i, cpt) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (\{i, no\}, cpt)$
- $(i, cpt) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, cpt) \circ (no, abs) \quad \rightsquigarrow \quad (i, ldup)$
- $(i, cpd) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, cpd)$
- $(i, cpd) \circ (no, cp) \quad \rightsquigarrow \quad (no, cp) \circ (i, cpd) \circ (i, cpd)$
- $(i, cpd) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, cpd) \circ (no, lbeta) \quad \rightsquigarrow \quad (no, lbeta) \circ (\{i, no\}, cpt)$

Lemma 7.2.12 Ein kompletter Satz Gabeldiagramme für die (cpd)- und (cpt)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad\rightsquigarrow\quad \xrightarrow{i,cpt} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cpt} \circ \xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad\rightsquigarrow\quad \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad\rightsquigarrow\quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,abs} \circ \xrightarrow{i,cpt} \quad\rightsquigarrow\quad \xrightarrow{i,ldup} \circ \xrightarrow{i,ldup} \circ \xleftarrow{no,abs} \circ \xleftarrow{no,abs}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,cpd} \quad\rightsquigarrow\quad \xrightarrow{i,cpd} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cp} \circ \xrightarrow{i,cpd} \quad\rightsquigarrow\quad \xrightarrow{i,cpd} \circ \xrightarrow{no,cpd} \circ \xleftarrow{no,cp}$
- $\xleftarrow{no,cpt} \circ \xleftarrow{no,lbeta} \circ \xrightarrow{i,cpd} \quad\rightsquigarrow\quad \xleftarrow{no,lbeta}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad\rightsquigarrow\quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,lbeta} \circ \xrightarrow{i,cpd} \quad\rightsquigarrow\quad \xrightarrow{i,cpt} \circ \xleftarrow{no,lbeta}$

Validierung der (cp)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert bei kleinen Termklassen keine Auffälligkeiten. Da wir aber davon ausgehen, daß die $(ldup)$-Transformation nicht korrekt ist und das Diagramm $((i, cpt) \circ (no, abs) \rightsquigarrow (i, ldup))$ auf jeden Fall benötigt wird, täuscht dieses Ergebnis die Vollständigkeit der Diagrammsätze vor.

```
$ jonah -RTS validate --ruleset=cp --redexes=ldup,lcom
--types=List --time --verbose --addrules --class=1,1,2
```

erzeugt die Ausgabe:

```
--rulesets=cp
-- 37,923 expressions in class (1, 2, 2, 0, 0).
-- 37,923 expressions tested.
-- rule set   |      tested expressions |  not found
-----------------------------------------------------
-- cp         |     14,676 |     38.6 % |          0
Execution time: 1 m, 0 s
```

7.2.7 Reduktionsdiagramme für (ucp)

Definition 7.2.6 Seien C, D beliebige Kontexte. Die (ucp)-Reduktion[7] ist definiert durch:

[7] unique copy

$C[\ \mathtt{let}\ x = s\ \mathtt{in}\ D[\ x\ ]\ ] \xrightarrow{ucp} D[\ s\ ]$
falls x nur *einmal* in $D[x]$, jedoch nicht innerhalb einer Abstraktion oder einer Konstruktoranwendung, vorkommt.

Markierung von (*ucp*)

Um einen (*ucp*)-Redex zu markieren, müssen wir prüfen wie of die Variable des `let`-Ausdrucks im Term `e` frei vorkommt, d. h. auch innerhalb von Abstraktionen. Falls die Variable genau einmal zu finden ist, kann der `let`-Ausdruck markiert werden, diesmal jedoch nur, wenn das Kopierziel sich in einem Oberflächenkontext befindet.

```
markUCP (ELet _ rec@False bs e)
    = [ ELet (Redex (UCP var)) rec bs e'
      | var <- cpvars
      , e'  <- markContextS (markVar var) e
      ]
      where
        cpvars   = [v | (v,e) <- bs, unique v]
        unique v = length (filter (== v) (getFreeVars e)) == 1
```

Reduktion von (*ucp*)

Da die (*ucp*)-Reduktion dem Ersetzen (*inlining*) eines Teilterms entspricht, reicht es hier die markierte Variable durch den Term zu ersetzen. Das umgebende `let` entfällt dabei.

```
reduceCoreExpr False level (ELet (Redex (UCP v)) False [(bv,be)] e)
    = [replaceMarkedVar False (show level) bv be e]
```

Diagrammsätze

Lemma 7.2.13 Ein kompletter Satz Vertauschungsdiagramme für die (*ucp*)-Reduktion enthält folgende Diagramme:

- $(i, ucp) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, ucp)$
- $(i, ucp) \circ (no, \{llet, lapp, lcase\}^*) \quad \rightsquigarrow \quad (no, \{llet, lapp, lcase\}^*) \circ (i, ucp)$
- $(i, ucp) \quad \rightsquigarrow \quad (no, llet) \circ (i, ucp) \circ (i, ldup) \circ (i, ldel)$
- $(i, ucp) \quad \rightsquigarrow \quad (no, a) \circ (i, ldel)$
  mit $a \in \{case, ndl, ndr\}$

Lemma 7.2.14 Ein kompletter Satz Gabeldiagramme für die (*ucp*)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ucp} \circ \xleftarrow{no,a}$

- $\xleftarrow{no,\{llet,lapp,lcase\}^*} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ucp} \circ \xleftarrow{no,\{llet,lapp,lcase\}^*}$
- $\xleftarrow{no,llet} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ucp} \circ \xrightarrow{i,ldup} \circ \xrightarrow{i,ldel}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ldel}$
  mit $a \in \{case, ndl, ndr\}$

Validierung der (*ucp*)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert bei kleinen Termklassen — wie schon (*ldup*) — einige nicht zu umgehende Fehler:

```
$ jonah -RTS validate --ruleset=ucp --redexes=all
--types=List --time --verbose --addrules --class=1,1,2
```

erzeugt die Ausgabe:

```
-- 37,923 expressions in class (1, 2, 2, 0, 0).
-- ucp: (let a=Nil in (let b=(Cons a) in b))
> i.ucp n.abs n.llet ==>
```

Jonah kann für diesen Term nur das angegebene Diagramm finden, welches aber leider für die Beweise nicht zu gebrauchen ist. Die NO-Reduktionen holen das in die Konstruktoranwendung kopierte `Nil` wieder heraus.

Die beiden folgenden Diagramme könnte dagegen wieder benutzt werden.

```
-- ucp: (let a=(Cons Nil) in (a Nil))
> i.ucp n.abs ==> n.abs n.llet i.ucp

-- ucp: (let a=Nil in (let b=Nil in ((Cons a) Nil)))
> i.ucp n.abs ==> i.lcom

-- ucp: no rule for ‘(let a=(Cons Nil) in (let b=Nil in (a b)))‘
-- ERROR: «ucp/a:(let a=(Cons Nil) in (let b=Nil in ([a] b)))»
      --> (let b=Nil in ((Cons Nil) b))
        : No rule found!
```

Da auch hier wiederum ein Term gefunden wird, der nicht durch ein vorhandenes oder ein ermitteltes Diagramm geschlossen werden kann, kann die Korrektheit der (*ucp*)-Transformation nicht gezeigt werden.

```
[...]
-- ucp: (let a=(Cons Nil) in (let b=(a Nil) in b))
> i.ucp ==> n.abs i.ucp i.ucp
-- 37,923 expressions tested.
-- rule set   |      tested expressions |  not found
----------------------------------------------------
-- ucp        |     15,343 |     40.4 % |          8
Execution time: 25 s
```

7.2.8 Lambda-Lifting

Mittels *Lambda-Lifting* werden freie Variablen abstrahiert, d. h. ein Ausdruck mit freien Variablen wird in eine λ-Abstraktion geändert, die auf die freie Variable angewendet wird. In funktionalen Programmiersprachen wird dieses Verfahren dazu benutzt, Superkombinatoren zu erstellen ([PJ87]).

$$C[\ s[z]\ ] \xrightarrow{llift} C[\ (\lambda x.s[x])\ z\ ]$$

Um nun die Korrektheit dieser Transformation zu zeigen, reicht es zu zeigen, wie sie wieder rückgängig gemacht werden kann. Die Reduktionsfolge (*lbeta*) $\circ$ (*lcv*) $\circ$ (*ldel*) bewerkstelligt dies:

$$C[\ (\lambda x.s[x])\ z\ ] \xrightarrow{lbeta} C[\ \texttt{let}\ x\ \texttt{=}\ z\ \texttt{in}\ s[x]\ ] \xrightarrow{lcv} C[\ \texttt{let}\ x\ \texttt{=}\ z\ \texttt{in}\ s[z]\ ] \xrightarrow{ldel} C[\ s[z]\ ]$$

Wenn (*lbeta*), (*lcv*) und (*ldel*) korrekte Transformationen sind, folgt daraus, daß *Lambda-Lifting* ebenfalls eine korrekte Transformation ist.

Mit derselben Argumentation läßt sich auch zeigen, daß *Lambda-Lifting* im rekursiven Kalkül eine korrekte Transformation ist. Einzig (*ldel*) hätte Probleme bereiten können, da es für `letrec`-Ausdrücke anders definiert ist. Da aber durch die (*lbeta*)- und (*lcv*)-Reduktionen keine zyklische Deklaration für x entstehen kann, reicht die schwache Form von (*ldel*) aus (siehe Kapitel 7.3.1).

7.2.9 Standardisierung

Wir wollen zeigen, daß unsere in den Definitionen 3.3.1 und 3.3.2 beschriebene Auswertung in Normalordnung standardisierend ist. D. h. wenn eine Reduktion $s \xrightarrow{*} s_0$ zu einer WHNF s_0 existiert, dann muß es eine NO-Reduktion zu einer WHNF geben, also $s\Downarrow$. Dazu müssen wir Reduktionsdiagramme für die Reduktionen (*ndl*) und (*ndr*) in Oberflächenkontexten statt in Reduktionskontexten angeben.

Lemma 7.2.15 Eine interne Reduktion (*i,ndl*) bzw. (*i,ndr*) sei eine Reduktion in einem Oberflächenkontext. Ein vollständiges Vertauschungsdiagramm für (*ndl*) und (*ndr*) ist:

$$(i,a) \circ (no,b) \quad \rightsquigarrow \quad (no,b) \circ (\{no,i\},a) \text{mit } a \in \{ndl, ndr\}$$

Satz 7.2.7 Sei s ein geschlossener Ausdruck mit einer Reduktion zu einer WHNF s_0: $s \xrightarrow{*} s_0$. Dann existiert eine terminierende NO-Reduktion für s.

Beweis. Induktion über die Länge der Reduktionsfolge $s \xrightarrow{*} s_0$ beginnend mit s_0. Wir nutzen dazu alle Reduktionsdiagramme der Kalkül-Regeln sowie das Diagramm aus Lemma 7.2.15. □

7.3 Reduktionsdiagramme im $\lambda_{nd,rec}$-Kalkül

Wegen der Probleme im vorherigen Kapitel, werden wir in diesem Kapitel nur die Implementation der jeweiligen Transformationen (Reduktionen) und die möglicherweise unvollständigen Diagramsätze angeben.

7.3.1 Reduktionsdiagramme für (*ldel*)

Wir passen die Definition der (*ldel*)-Reduktion von Seite 99 für `letrec`-Ausdrücke an.

Definition 7.3.1 Sei C ein beliebiger Kontext. Die (*ldel*)-Reduktion ist definiert durch:

$C[\,\texttt{letrec}\ x = s\ \texttt{in}\ t\,] \xrightarrow{ldel} C[\,t\,]$,
falls x nicht in t vorkommt.

$C[\,\texttt{letrec}\ x_1 = s_1, \ldots, x_m = s_m\ \texttt{in}\ t\,] \xrightarrow{ldel} C[\,\texttt{letrec}\ x_j = s_j, \ldots, x_m = s_m\ \texttt{in}\ t\,]$,
falls alle x_i mit $i < j$ weder in den $s_j, \ldots, s_m$ noch in t vorkommen.

$C[\,\texttt{letrec}\ x_1 = s_1, \ldots, x_m = s_m\ \texttt{in}\ t\,] \xrightarrow{ldel} C[\,t\,]$,
falls alle x_i mit $1 \leq i \leq m$ nicht in t vorkommen.

Markierung von (*ldel*)

Im rekursiven Kalkül darf die zu löschende Variable weder in `expr` noch in einem der übrigen Bindungsterme vorkommen, da diese vielleicht noch in `t` referenziert sind.

```
markLDel (ELet _ rec@True bs expr)
    = [ ELet (Redex (LDel var)) rec bs expr
      | var <- [v  | (v,_) <- bs]
      , var `notElem` (concat (freevars:[f | (v,f) <- freebs, v /= var]))
      ]
      where
        freevars = getFreeVars expr
        freebs   = [(v,getFreeVars e) | (v,e) <- bs]
```

Die (*ldelcyc*)-Reduktion ist etwas kompliziert. Wir erzeugen aus der Bindungsliste eine Liste aller Variablen mit den freien Variablen ihres gebundenen Ausdrucks (`letvars`). Dann werden die freien Variablen des Ausdrucks `expr` ermittelt (`freevars`). Wir müssen nun alle Variablen finden, die weder in `expr` noch in den Bindungen frei vorkommen.

```
markLDelCyc1 (ELet _ False _ expr) = []
markLDelCyc1 (ELet _ True bs expr)
    = [ ELet (Redex (LDelCyc1 unused)) True bs expr
      | lenUnused > 0
      , length bs /= lenUnused
      ]
      where
        lenUnused = length unused
        letvars   = [ (v,fvs)
                    | (v,e) <- bs
                    , let fvs = filter (/= v) (getFreeVars e)
                    ]
        freevars  = getFreeVars expr
        unused    = collect letvars freevars
```

Dazu teilen wir mittels der Funktion `collect` die Menge der Variablen derart auf, daß alle Variablen, die in der Liste der „freien“ sind, herausgefiltert werden. Die freien Variablen der herausgefilterten Bindungen müssen allerdings zu der Liste der „freien“ hinzugefügt werden, und die Partitionierung wiederholt werden. Am Ende bleiben nur diejenigen Variablen übrig, die weder direkt noch indirekt im Ausdruck `expr` frei sind.

```
        collect lvs fv
          = case partition (\(v,_) -> v `elem` fv) lvs
              of ([],ys) -> [v | (v,_) <- ys]
                 (xs,ys) -> let fv' = concat [vs | (_,vs) <- xs] ++ fv
                            in collect ys fv'
markLDelCyc1 expr = []
```

Die nicht benötigten Variablen werden in einer Liste der Markierung `LDelCyc1` hinzugefügt, wenn sie nicht leer ist. Falls alle Variablenbindungen gelöscht werden können, dann wird der Ausdruck ebenfalls nicht als (*ldelcyc1*) markiert, da dazu die (*ldelcyc2*)-Reduktion dient.

```
markLDelCyc2 (ELet _ False _ expr) = []
markLDelCyc2 (ELet _ True bs expr)
    = [ ELet (Redex LDelCyc2) True bs expr
      | [] == (letvars `intersect` freevars)
      ]
      where
        freevars = getFreeVars expr
        letvars  = [v  | (v,_) <- bs]
markLDelCyc2 expr = []
```

Reduktion von (*ldel*)

Im rekursiven Kalkül wird die zu löschende Bindung entfernt. Wenn keine weiteren Bindungen im `letrec`-Ausdruck vorhanden ist, dann kann er auch entfernt werden.

```
reduceCoreExpr True level (ELet (Redex (LDel v)) r bs e)
    = case removeBinding v bs
        of []  -> [e]
           bs' -> [ELet Unmarked True bs' e]
```

Ein (*ldelcyc1*)-Redex wird reduziert, indem alle in `vs` befindlichen Variablen aus den Bindungen entfernt werden — bzw. alle Bindungen werden übernommen, die nicht in `vs` sind.

```
reduceCoreExpr True level (ELet (Redex (LDelCyc1 vs)) r bs e)
    = [ELet Unmarked r bs' e]
      where
        bs' = [b | b@(v,_) <- bs, v `notElem` vs]
```

Beim Reduzieren des (*ldelcyc2*)-Redex bleibt nur der Ausdruck `e` übrig.

```
reduceCoreExpr True level (ELet (Redex LDelCyc2) r bs e)
    = [e]
```

Diagrammsätze

Wir geben je einen kompletten Satz Vertauschungs- und Gabeldiagramme an, die für den Beweis der kontextuellen Äquivalenz der (*ldel*)-Reduktion benötigt werden.

Lemma 7.3.1 Ein kompletter Satz Vertauschungsdiagramme für die (*ldel*)-Reduktion enthält folgende Diagramme:

- $(i, ldel) \circ (no, a) \quad \leadsto \quad (no, a) \circ (i, ldel)$
- $(i, ldel) \circ (no, a) \quad \leadsto \quad (no, \{lapp, llet, lcase\}^{+}) \circ (no, a) \circ (i, ldel)$
- $(i, ldel) \circ (no, \{lapp, llet, lcase\}^{+}) \quad \leadsto \quad (no, \{lapp, llet, lcase\}^{*}) \circ (i, ldel)$

Lemma 7.3.2 Ein kompletter Satz Gabeldiagramme für die (*ldel*)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,ldel} \quad \leadsto \quad \xrightarrow{i,ldel} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xleftarrow{no,\{lapp,llet,lcase\}^{+}} \circ \xrightarrow{i,ldel} \quad \leadsto \quad \xrightarrow{i,ldel} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,\{lapp,llet,lcase\}^{+}} \circ \xrightarrow{i,ldel} \quad \leadsto \quad \xrightarrow{i,ldel} \circ \xleftarrow{no,\{lapp,llet,lcase\}^{*}}$

7.3.2 Reduktionsdiagramme für (*lcv*)

Wir passen die Definition 7.2.2 an den rekursiven Kalkül an.

Definition 7.3.2 Seien C, D, D' beliebige Kontexte und S ein Oberflächenkontext. Die (*lcvt*)- und (*lcvd*)-Reduktion ist definiert durch:

$$C[\ \texttt{let}\ x = y, \ldots\ \texttt{in}\ S[\ x\ ]\ ] \xrightarrow{lcvt} C[\ \texttt{letrec}\ x = y, \ldots\ \texttt{in}\ S[\ y\ ]\ ]$$

$$C[\ \texttt{let}\ x_1 = y, x_2 = S[\ x_1\ ], \ldots\ \texttt{in}\ t\ ] \xrightarrow{lcvt} C[\ \texttt{let}\ x_1 = y, x_2 = S[\ y\ ], \ldots\ \texttt{in}\ t\ ]$$

$$C[\ \texttt{let}\ x = y, \ldots\ \texttt{in}\ D[\ \lambda z . D'[x]\ ]\ ] \xrightarrow{lcvd} C[\ \texttt{let}\ x = y, \ldots\ \texttt{in}\ D[\ \lambda z . D'[y]\ ]\ ]$$

$$C[\ \texttt{let}\ x_1 = y, x_2 = D[\ \lambda z . D'[x_1]\ ], \ldots\ \texttt{in}\ t\ ] \xrightarrow{lcvd} C[\ \texttt{let}\ x_1 = y, x_2 = D[\ \lambda z . D'[y]\ ], \ldots\ \texttt{in}\ t\ ]$$

Markierung von (*lcvt*) und (*lcvd*)

Wir verallgemeinern die in Kapitel 7.2.3 beschriebenen Funktionen.

```
markLCVT (ELet _ rec bs expr)
    = [ ELet (Redex (LCVT var)) rec bs expr'
      | var   <- cpvars
      , expr' <- markContextsS (markVar var) expr
      ]
      ++
      [ ELet (Redex (LCVT var)) rec bs' expr
      | rec
      , var     <- cpvars
      , (v,e')  <- bs
      , e       <- markContextsS (markVar var) e'
      , let bs' = updateBinding v e bs
      ]
      where
        cpvars = [ v
                 | (v,e) <- bs
                 , isVariable e
                 , v /= getVariableName e
                 ]
markLCVT expr = []
```

Die gleichen Änderungen müssen an der Markierungsfunktion für (*lcvd*) gemacht werden.

```
markLCVD (ELet _ rec bs expr)
    = [ ELet (Redex (LCVD var)) rec bs expr'
      | var   <- cpvars
      , expr' <- markContextC (markVar var) (True,True) expr
                 \\
                 markContextsS (markVar var) expr
      ]
      ++
      [ ELet (Redex (LCVD var)) rec bs' expr
      | rec
      , var     <- cpvars
      , (v,e')  <- bs
      , e       <- markContextC (markVar var) (True,True) e'
                   \\
                   markContextsS (markVar var) e'
      , let bs' = updateBinding v e bs
      ]
      where
        cpvars = [ v
                   | (v,e) <- bs
                   , isVariable e
                   , v /= getVariableName e
                   ]
markLCVD expr = []
```

Reduktion von (*lcvt*) und (*lcvd*)

Die Reduktion ist von der Umstellung auf den rekursiven Kalkül nicht betroffen, da sie durch die (cp)-Reduktion realisiert wird (siehe Kapitel 7.2.3, Seite 102).

Diagrammsätze

Lemma 7.3.3 Ein kompletter Satz Vertauschungsdiagramme für die (lcv)-Reduktion enthält folgende Diagramme:

- $(i, lcvt) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, lcvt)$
- $(i, lcvt) \circ (no, cp) \quad \rightsquigarrow \quad (no, cp) \circ (no, cp) \circ \xleftarrow{i,cpt}$
- $(i, lcvt) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, lcvd) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, lcvd)$
- $(i, lcvd) \circ (no, cp) \quad \rightsquigarrow \quad (no, cp) \circ (i, lcvd) \circ (i, lcvd)$
- $(i, lcvd) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, lcvd) \circ (no, lbeta) \quad \rightsquigarrow \quad (no, lbeta) \circ (i, lcvt)$

Lemma 7.3.4 Ein kompletter Satz Gabeldiagramme für die ($lcom$)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,lcvt} \quad \rightsquigarrow \quad \xrightarrow{i,lcvt} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,lcvt} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,cp} \circ \xleftarrow{no,cp} \circ \xrightarrow{i,lcvt} \quad \rightsquigarrow \quad \xleftarrow{i,cpt} \circ \xleftarrow{no,cp}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,lcvd} \quad \rightsquigarrow \quad \xrightarrow{i,lcvd} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cp} \circ \xrightarrow{i,lcvd} \quad \rightsquigarrow \quad \xrightarrow{i,lcvd} \circ \xrightarrow{i,lcvd} \circ \xleftarrow{no,cp}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,lcvd} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,lbeta} \circ \xrightarrow{i,lcvd} \quad \rightsquigarrow \quad \xrightarrow{i,lcvt} \circ \xleftarrow{no,lbeta}$

7.3.3 Reduktionsdiagramme für (*ldup*)

Definition 7.3.3 Seien C, D beliebiger Kontexte. Die *(ldup)-Reduktion* ist definiert durch:

$C[$ `letrec` x `=` t`, ... in` $D[\ x, ..., x\ ]$ $\xrightarrow{ldup}$ $C[$ `letrec` x `=` t, y `=` t`, ... in` $D[\ z_1, ..., z_n]\ ]$,
falls t kopierbar, $z_i \in \{x, y\}$ und y neu.

Es werden in allen Kontexten alle möglichen Variablenbelegungen (x oder y) erzeugt.

Markierung von (*ldup*)

Da die Markierung eines (*ldup*)-Redex sich im rekursiven Fall gleich bleibt, müssen wir nur die Reduktionsfunktion von Seite 105 abändern.

Reduktion von (*ldup*)

Wir unterscheiden nun beide Kalküle, wobei im rekursiven Kalkül die neue Bindung sofort in den `letrec`-Ausdruck übernommen wird und dann nur noch die Variablenbelegungen generiert werden müssen.

```
reduceCoreExpr rec level (ELet (Redex (LDup v)) r bs e)
    = if rec
         then [ ELet Unmarked True (bs ++ bs') e'
              | e' <- e:(setv' [e])
              ]
         else [ ELet Unmarked False bs e'
              | e' <- markContextS insertLet e
              ]
         where
```

Der Rest der Funktion bleibt unverändert (siehe Kapitel 7.2.5, Seite 105).

Diagrammsätze

Lemma 7.3.5 Ein kompletter Satz Vertauschungsdiagramme für die (*ldup*)-Reduktion enthält folgende Diagramme:

- $(i, ldup) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, ldup)$
- $(i, ldup) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, ldup) \circ (no, \{llet, lapp, lcase\}^+) \quad \rightsquigarrow \quad (no, \{llet, lapp, lcase\}^*) \circ (i, ldup)$

Lemma 7.3.6 Ein kompletter Satz Gabeldiagramme für die (*ldup*)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,ldup} \quad \rightsquigarrow \quad \xrightarrow{i,ldup} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,ldup} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,\{llet,lapp,lcase\}^*} \circ \xrightarrow{i,ldup} \quad \rightsquigarrow \quad \xrightarrow{i,ldup} \circ \xleftarrow{no,\{llet,lapp,lcase\}^*}$

7.3.4 Reduktionsdiagramme für (cp)

Wir definieren die (cp)-Reduktionen neu:

Definition 7.3.4 Seien C, D, D' beliebige Kontexte, c ein Konstruktor und t ein kopierbarer Term (Abstraktion oder Konstruktoranwendung). Die (cpd)- und (cpt)-Reduktion sind definiert durch:

$$
\begin{array}{lcl}
C[\ \texttt{letrec}\ x\ \texttt{=}\ t\texttt{,}\ \ldots\ \texttt{in}\ D[\ \lambda y\,.\,D'[x]\ ]\ ] & \xrightarrow{cpd} & C[\ \texttt{letrec}\ x\ \texttt{=}\ t\texttt{,}\ \ldots\ \texttt{in}\ D[\ \lambda y\,.\,D'[t]\ ]\ ] \\
C[\ \texttt{letrec}\ x_1\ \texttt{=}\ t\texttt{,}\ x_2\ \texttt{=}\ D[\ \lambda y\,.\,D'[x_1]\ ]\ \texttt{in}\ s\ ] & \xrightarrow{cpd} & C[\ \texttt{letrec}\ x_1\texttt{=}t\texttt{,}\ x_2\texttt{=}D[\lambda y\,.\,D'[t]]\ \texttt{in}\ s] \\
C[\ \texttt{letrec}\ x\ \texttt{=}\ t\texttt{,}\ \ldots\ \texttt{in}\ S[x]\ ] & \xrightarrow{cpt} & C[\ \texttt{letrec}\ x\ \texttt{=}\ t\texttt{,}\ \ldots\ \texttt{in}\ S[t]\ ] \\
C[\ \texttt{letrec}\ x_1\ \texttt{=}\ t\texttt{,}\ x_2\ \texttt{=}\ S[x_1]\texttt{,}\ \ldots\ \texttt{in}\ s\ ] & \xrightarrow{cpd} & C[\ \texttt{letrec}\ x_1\ \texttt{=}\ t\texttt{,}\ x_2\ \texttt{=}\ S[t]\texttt{,}\ \ldots\ \texttt{in}\ s\ ]
\end{array}
$$

Die Markierung der beiden Redex-Typen ist ausführlich in Kapitel 3.3.3 auf Seite 47 beschrieben.

Die (cpt)- und (cpd)-Reduktionen können wie ein normales (cp) reduziert werden. Wir müssen sie nur bei der Validierung unterscheiden. Zur Reduktion ändern wir einfach nur die (cpt)- bzw. (cpd)-Markierung zu (cp) und reduzieren als (cp).

```
reduceCoreExpr rec level (ELet (Redex (CopyD v)) r bs e)
    = reduceCoreExpr rec level (ELet (Redex (Copy v)) r bs e)

reduceCoreExpr rec level (ELet (Redex (CopyT v)) r bs e)
    = reduceCoreExpr rec level (ELet (Redex (Copy v)) r bs e)
```

Diagrammsätze

Die Diagrammsätze sind die gleichen wie in Kapitel 7.2.6.

7.3.5 Reduktionsdiagramme für (ucp)

Die in Definition 7.2.6 eingeführte (ucp)-Reduktion muß für Kalküle mit `letrec`-Ausdrücken erweitert werden, da sich das Kopierziel auch in den Bindungen befinden kann und überprüft werden muß, ob der kopierte Ausdruck keine rekursive Referenz auf sich selbst enthält.

Definition 7.3.5 Seien C beliebige Kontexte und S Oberflächenkontexte. Die (ucp)-Reduktion ist definiert durch:

$C[\,\texttt{letrec}\ x = s\ \texttt{in}\ S[\,x\,]\,] \xrightarrow{ucp} C[\,S[\,s\,]\,]$
falls x *genau einmal* in $S[x]$ und nicht in s vorkommt, wobei s beliebig ist.

$C[\,\texttt{letrec}\ x_1 = s_1, x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ S[\,x\,]\,]$
$\xrightarrow{ucp} C[\,\texttt{letrec}\ x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ S[\,s\,]\,]$
falls x *genau einmal* in $S[x]$ und nicht in den s_i vorkommt, wobei s_1 beliebig ist.

$C[\,\texttt{letrec}\ x_1 = s_1, y = S[\,x\,], x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ t\,]$
$\xrightarrow{ucp} C[\,\texttt{letrec}\ y = S[\,s\,], x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ t\,]$
falls x *genau einmal* in $S[x]$, weder in den s_i noch in t vorkommt, und s beliebig ist.

Falls s bzw. s_1 eine Abstraktion ist, dann verhält sich (ucp) wie eine (cp)- mit nachfolgender $(ldel)$-Reduktion.

Implementierung von (ucp)

Zur Markierung von (ucp)-Redexen werden zuerst alle `letrec`-gebundenen Variablen gesammelt, die genau einmal im gesamten `letrec`-Ausdruck vorkommen. Dazu zählen wir die Vorkommen der Variable in allen Teiltermen[8], bilden daraus Summe und prüfen, ob diese 1 ergibt. Dann werden diese Variablen in allen Oberflächenkontexten von `e` und den gebundenen Ausdrücken markiert.

```
markUCP (ELet _ rec@True bs e)
    = [ ELet (Redex (UCP var)) rec bs e'
      | var <- cpvars
      , e'  <- markContextS (markVar var) e
      ]
      ++
      [ ELet (Redex (UCP var)) rec bs' e
      | var     <- cpvars
      , (bv,be) <- bs
      , bv /= var
      , be'     <- markContextS (markVar var) be
      , let bs'  = updateBinding bv be' bs
      ]
      where
        cpvars    = [v | (v,e) <- bs, unique v]
        count v e = length (filter (== v) (getFreeVars e))
        unique v  = (count v e + sum [count v be | (bv,be) <- bs]) == 1
markUCP expr = []
```

Das Reduzieren eines (ucp)-Redex gestaltet sich da schon einfacher, da der Ausdruck direkt an die Stelle der markierten Zielvariable `v` kopiert werden kann. Eine Umbenennung der Kopie

[8] Die Funktion `getFreeVars` liefert alle freien Variablen eines Ausdrucks, wobei Mehrfachvorkommen berücksichtigt werden.

ist nicht nötig, da durch das nachfolgende Löschen der Bindung, unsere Variablennamenkonvention nicht verletzt wird. Zuletzt muß noch geprüft werden, ob die Liste der Bindungen leer ist, so daß der umgebende `letrec`-Ausdruck entfernt werden kann.

```
reduceCoreExpr True level (ELet (Redex (UCP v)) True bs e)
    = case bs'
        of []        -> [e']
           otherwise -> [ELet Unmarked True bs' e']
      where
        ve  = lookupBinding v bs
        e'  = replaceMarkedVar False "" v ve e
        bs' = [ (bv,be')
              | (bv,be) <- bs
              , bv /= v
              , let be' = replaceMarkedVar False "" v ve be
              ]
```

Reduktionsdiagramme von (*ucp*)

Lemma 7.3.7 Ein kompletter Satz Vertauschungsdiagramme für die (ucp)-Reduktion enthält folgende Diagramme:

- $(i, ucp) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, ucp)$
- $(i, ucp) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, ldel)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, ucp) \circ (no, \{llet, lapp, lcase\}^*) \quad \rightsquigarrow \quad (no, \{llet, lapp, lcase\}^*) \circ (i, ucp)$
- $(i, ucp) \quad \rightsquigarrow \quad (no, cp) \circ (i, ldel)$

Lemma 7.3.8 Ein kompletter Satz Gabeldiagramme für die (ucp)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ucp} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ldel} \circ \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,\{llet,lapp,lcase\}^*} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ucp} \circ \xleftarrow{no,\{llet,lapp,lcase\}^*}$
- $\xleftarrow{no,cp} \circ \xrightarrow{i,ucp} \quad \rightsquigarrow \quad \xrightarrow{i,ldel}$

7.4 Reduktionsdiagramme im $\lambda_{nd,rec,eql}$-Kalkül

Wegen der Schwierigkeiten bei der Angabe von Reduktionsdiagrammen für die (cp)- und $(ldup)$-Reduktionen, versuchen wir diese Probleme dadurch zu umgehen, indem wir den Basis-Kalkül etwas ändern. Genauer gesagt wird das die (cp)-Reduktion sein. Wir werden wiederum sehen, daß diese Änderung nicht ausreicht, und wir eine Anpassung der (*case*)-Reduktion vornehmen müssen. Daraus ergeben sich zwei verschiedene Kalküle ($\lambda_{nd,rec,eql}$ und $\lambda_{nd,rec,case}$). Wir stellen in diesem Kapitel den ersten und im nächsten Kapitel den zweiten vor.

Statt Variablen zu kopieren, kann natürlich auch gleich der Ausdruck, der an die zu kopierende Variable gebunden ist, kopiert werden. Da dieser ebenfalls eine Variable sein kann, müssen also alle Bindungen „rückwärts" verfolgt werden, bis diejenige Variable gefunden wird, die keine Variable mehr an sich bindet. Der an diese Variable gebundene Ausdruck kann dann direkt kopiert werden.

Wir verändern also den Kalkül $\lambda_{nd,rec}$ dahingehend, daß keine Variablen kopiert werden, sondern ein kopierbarer Term am Ende einer Folge von Indirektionen. Z. B. wird im Term ($\texttt{letrec}\ x = \texttt{Nil},\ y = x\ \texttt{in}\ y$) das `Nil` direkt an die Stelle des y kopiert.

Zusätzlich werden wir zwei neue Programmtransformationen vorstellen, eine Variante der (*ldel*)-Reduktion und die (*eql*)-Reduktion, die gleiche Bindungen identifiziert und die Variablen im Term austauscht. Beide Transformationen werden für die Reduktionsdiagrammen der (*cp*)-Reduktion benötigt.

7.4.1 Verfolgen von Indirektionen in (*cp*)

Wir ändern die (*cp*)- und erweitern die (*case*)-Reduktion in der Kalkül-Definition 3.1.10 auf Seite 24. Die (*abs*)-Reduktion wird entfernt, da die Konstruktorargumente während der (*case*)-Reduktion abstrahiert werden.

Definition 7.4.1 Der *nichtdeterministische Kalkül* $\lambda_{nd,rec,case}$ enthält die Reduktionsregeln aus Definition 3.1.10, mit Ausnahme der (*abs*)-Reduktion. Die Reduktionsregeln für (*cp*) und (*case*) werden folgendermaßen geändert:

$$C[\ \texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_i = x_{i-1}, x_{i+1} = s_{i+1}, \ldots\ \texttt{in}\ D[\ x_i\ ]\ ]$$
$$\xrightarrow{cp}\quad C[\ \texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_i = x_{i-1}, x_{i+1} = s_{i+1}, \ldots\ \texttt{in}\ D[\ s_1\ ]\ ]$$
falls s_i eine Abstraktion oder eine pure Konstruktoranwendung ist.

$$C[\ \texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_i = x_{i-1}, x_{i+1} = D[x_i], \ldots\ \texttt{in}\ s\ ]$$
$$\xrightarrow{cp}\quad C[\ \texttt{letrec}\ x_1 = s_1, x_2 = x_1, \ldots, x_i = x_{i-1}, x_{i+1} = D[s_1], \ldots\ \texttt{in}\ s\ ]$$
falls s_i eine Abstraktion oder eine pure Konstruktoranwendung ist.

Die Markierung des NO-Redex geht genau so vonstatten wie in Definition 3.3.2 auf Seite 36. Nur die Erkennung eines (*cp*)-Redex ist neu, die übrigen Teile des Algorithmus bleiben erhalten.

Definition 7.4.2 (MarkNO$_{rec,case}$) Der *NO-Redex* eines $\Lambda_{nd,rec}$-Ausdrucks t wird mittels einer Menge von Regeln definiert, die die Markierungen E (*evaluation*) oder H (*halt*) bis zu einem finalen Teilausdruck in t auf- und abbewegen. Der Algorithmus beginnt mit dem unmarkierten Ausdruck t^E:

1. Erkennung eines NO-Redex:

(cp) $R[$ `letrec` x_1 `=` s_1`,`x_2 `=` x_1`,`...`,`x_i `=` x_{i-1}`,`x_{i+1} `=` s_{i+1}`,`... `in` $D[\ x_i\ ]\ ]$
falls $s_i = (\backslash v\,.\,t)$ oder $s_i = (c_{A,i}\ t_1 \ldots t_n)$

(cp) $R[$ `letrec` x_1 `=` s_1`,`x_2 `=` x_1`,`...`,`x_i `=` x_{i-1}`,`x_{i+1} `=` $D[x_i]$`,`... `in` $s\ ]$
falls $s_i = (\backslash v\,.\,t)$ oder $s_i = (c_{A,i}\ t_1 \ldots t_n)$

2. Erkennung einer WHNF:
   siehe Definition 3.3.2, Seite 36

3. Erkennung eines irreduziblen Ausdrucks:
   siehe Definition 3.3.2, Seite 36

4. Fortsetzung des Markierungsverfahrens, falls keine der obigen Regeln angewandt werden kann:
   siehe Definition 3.3.2, Seite 36

   Das Markierungsverfahren stoppt, falls entweder

   (a) ein NO-Redex oder
   (b) ein irreduzibler Ausdruck oder
   (c) eine WHNF, CWHNF oder VWHNF

   entdeckt wird.

Markierung von (*cp*)

Für die Erkennung eines (cp)-Redex muß nur die Markierung eines `letrec`-Ausdrucks geändert werden. Da nur wenige Änderungen in die Funktion einfließen, halten wir die Original-Passagen in grau.

```
markNORedex True expr@(ELet m True bs e)
    = case markNORedex True e
        of HasError s    e' -> HasError s  expr
           HasRedex r    e' -> HasRedex r  (ELet m    True bs e')
           HasWHNF       e' -> HasWHNF     (ELet WHNF True bs e )
           HasCWHNF ta a e' -> HasCWHNF ta a expr
           HasVWHNF v    e' -> if v `notElem` boundVars
                                  then HasVWHNF v (ELet m True bs e')
                                  else checkBindings v v "" [v] e'
        where
```

Der Markierungsalgorithmus muß nun alle Bindungen derart untersuchen, daß bei jedem Auftreten einer VWHNF geprüft wird, ob die Variable `v` in dem `letrec`-Ausdruck deklariert ist. Die Bindungen werden so lange durchsucht bis alle Variablen-Bindungen genau einmal betrachtet wurden. Die Variable muß nicht unbedingt in dem betrachteten `letrec`-Ausdruck deklariert sein. Wenn allerdings alle Bindungen untersucht worden sind und keine andere Markierung — ein Redex oder ein Fehler — möglich ist, dann kann nur noch eine zyklische Folge von Variablenreferenzen (inkl. Indirektionen) vorhanden sein, die zu einem Fehler führt.

```
boundVars
  = [x | (x,_) <- bs]

hasRedex red mark bs' e'
  = HasRedex red (ELet mark True bs' e')

newbs var e'
  = updateBinding var e' bs

cycleError
  = HasError ("Reduction.markNORedex (Letrec): "
              ++ "cyclic term in expression: "
              ++ show expr) expr
```

In der Funktion `checkBindings` wird zusätzlich überprüft, ob die Bindung eine Variable ist, also ein Verweis auf einen weiteren Ausdruck oder wiederum eine Variable. In diesem Fall muß über alle Indirektionen hinweg weitergeprüft werden.

```
checkBindings copyVar refVar tVar visVars e'
  = case vexpr
      of EVar _ iv    -> checkIndirections iv
         ELam _ _ _   -> copyRedex
         ELet _ _ _ _ -> lletrec2Redex
         otherwise    -> checkBoundExprs
      where

        vexpr
          = lookupBinding refVar bs

        copyRedex
          = if copyVar == refVar
               then hasRedex (Copy copyVar) (Redex (Copy copyVar)) bs e'
               else hasRedex (Copy refVar ) (Redex (Copy refVar ))
                                            (newbs tVar e') e

        lletrec2Redex
          = hasRedex (LLetrec2 refVar) (Redex (LLetrec2 refVar)) bs e
```

Wenn die Bindung eine Indirektion ist, dann können wir direkt mit der Suche fortfahren, solange noch unbesuchte Bindungen vorhanden sind.

```
        checkIndirections iv
          = if iv `elem` visVars
               then cycleError
               else checkBindings iv iv refVar (refVar:visVars) e'
```

Ansonsten wird der `letrec`-gebundene Term markiert. Im Falle einer VWHNF wird überprüft, ob eine zyklische Deklaration der Variablen vorliegt.

```
        checkBoundExprs
          = case markNORedex True vexpr
              of HasError s  ve -> HasError s  e
                 HasRedex r' ve -> hasRedex r' m (newbs refVar ve) e
                 HasCWHNF _ _ _ -> copyRedex
                 HasVWHNF v' ve -> checkCycles v' ve
```

Die Überprüfung auf Zyklen ist wiederum dieselbe wie im Original-Code.

```
checkCycles v' ve
  = if v' `elem` visVars
       then cycleError
       else if v' `elem` boundVars
               then checkBindings copyVar v' refVar (v':visVars) ve
               else HasVWHNF v' (ELet m True (newbs refVar ve) e)
```

Diagrammsätze für (cp)

Lemma 7.4.1 Ein kompletter Satz Vertauschungsdiagramme für die (cpd)- und (cpt)-Reduktion enthält folgende Diagramme:

- $(i, cpt) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (\{i, no\}, cpt)$
- $(i, cpt) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, cpt) \circ (no, abs) \quad \rightsquigarrow \quad \xleftarrow{i,ldels} \circ (i, eql)$
- $(i, cpd) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, cpd)$
- $(i, cpd) \circ (no, cp) \quad \rightsquigarrow \quad (no, cp) \circ (i, cpd) \circ (i, cpd)$
- $(i, cpd) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, cpd) \circ (no, lbeta) \quad \rightsquigarrow \quad (no, lbeta) \circ (\{i, no\}, cpt)$

Die Diagrammsätze unterscheiden sich von denen aus Kapitel 7.2.6 nur durch das dritte Diagramm. Als Ersatz der $(ldup)$-Reduktion wird hier $(ldels)$ und (eql) benutzt.

Wir werden jedoch in den nächsten Abschnitten sehen, daß auch für diese Transformationen, speziell für $(ldels)$, keine vollständigen Diagrammsätze gefunden werden können.

7.4.2 Reduktionsdiagramme für (eql)

Sind zwei Terme in den Bindungen eines `letrec`-Ausdrucks gleich, dann sind die Variablen innerhalb des Terms austauschbar. Auch wenn eine nicht-rekursive Variante nicht benötigt wird, so soll sie doch erwähnt sein.

Definition 7.4.3 Seien C,D beliebige Kontexte und R Reduktionskontexte. Die (eql)-Reduktion (*equal let*) ist definiert durch:

$C[\ \texttt{let}\ x_1\ \texttt{=}\ s\ \texttt{in}\ R[\ \texttt{let}\ x_2\ \texttt{=}\ s'\ \texttt{in}\ E[\ x_1\ ]\ ]\ ]$
$\xrightarrow{eql}\quad C[\ \texttt{let}\ x_1\ \texttt{=}\ s\ \texttt{in}\ R[\ \texttt{let}\ x_2\ \texttt{=}\ s'\ \texttt{in}\ E[\ x_2\ ]\ ]\ ]$

bzw.

$C[\ \texttt{letrec}\ x_1\ \texttt{=}\ s\texttt{,}x_2\ \texttt{=}\ s'\texttt{,}\ldots\ \texttt{in}\ D[\ x_1\ ]\ ]$
$\xrightarrow{eql}\quad C[\ \texttt{letrec}\ x_1\ \texttt{=}\ s\texttt{,}x_2\ \texttt{=}\ s'\texttt{,}\ldots\ \texttt{in}\ D[\ x_2\ ]\ ]$

$C[\ \texttt{letrec}\ x_1\ \texttt{=}\ s\texttt{,}x_2\ \texttt{=}\ s'\texttt{,}\ldots\texttt{,}x_i\ \texttt{=}\ D[\ x_1\ ]\texttt{,}\ldots\ \texttt{in}\ t\ ]$
$\xrightarrow{eql}\quad C[\ \texttt{letrec}\ x_1\ \texttt{=}\ s\texttt{,}x_2\ \texttt{=}\ s'\texttt{,}\ldots\texttt{,}x_i\ \texttt{=}\ D[\ x_2\ ]\texttt{,}\ldots\ \texttt{in}\ t\ ]$

falls $s \equiv s'$, also Gleichheit bis auf Umbenennung der Variablen.

Markierung von (*eql*)

```
markEQL (ELet _ rec@False bs1@[(v1,e1)] expr)
    = [ ELet (Redex (EQL v1)) rec bs1 expr'
      | expr' <- markContextR markEqual expr
      ]
      where
        markEqual (ELet m rec2 bs2@[(v2,e2)] e')
          = [ ELet m rec2 bs2 e''
            | e1 == e2
            , e'' <- markContextC (markVar v2) (True,True) e'
            ]
        markEqual x = []
```

Zur Markierung eines (*eql*)-Redex markieren wir nacheinander alle Bindungsvariablen `v1` im Ausdruck `expr` und prüfen, ob der gebundene Term noch einmal an eine andere Variable gebunden ist. In diesem Fall markieren wir den `letrec`-Ausdruck als (*eql*)-Redex und führen die Variable `v2` mit.

```
markEQL (ELet _ rec@True bs expr)
    = [ ELet (Redex (EQL v2)) rec bs expr'
      | (v1,ve1) <- bs
      , expr'    <- markContextC (markVar v1) (True,True) expr
      , v2       <- [v | (v,e) <- bs, v /= v1, e == ve1]
      ]
```

In den Bindungen wird ebenso verfahren:

```
      ++
      [ ELet (Redex (EQL v2)) rec bs' expr
      | (v0,ve0) <- bs
      , (v1,ve1) <- [b | b@(v,e) <- bs, v /= v0]
      , ve0'     <- markContextC (markVar v1) (True,True) ve0
      , v2       <- [v | (v,e) <- bs, v /= v0, v /= v1, e == ve1]
      , let bs' = updateBinding v0 ve0' bs
      ]
markEQL expr = []
```

Reduktion von (*eql*)

Im Ausdruck ist eine Variable markiert und soll durch die Variable `v` ersetzt werden. Eine der Bindungsvariablen ist in `expr` oder den Bindungen markiert, und wird mittels `replaceMarkedVar` durch die Variable `v` ersetzt.

```
reduceCoreExpr rec level (ELet (Redex (EQL v)) r bs e)
    = [ELet Unmarked rec bs' e']
      where
        v'  = EVar Unmarked v
        e'  = replaceMarkedVar True "" "dummy" v' e
        bs' = [ (v1,e1')
              | (v1,e1) <- bs
              , let e1' = replaceMarkedVar True "" "dummy" v' e1
              ]
```

Diagrammsätze

Lemma 7.4.2 Ein kompletter Satz Vertauschungsdiagramme für die (*eql*)-Reduktion enthält folgende Diagramme:

- $(i, eql) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, eql)^*$

Lemma 7.4.3 Ein kompletter Satz Gabeldiagramme für die (*eql*)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,eql} \quad \rightsquigarrow \quad \xrightarrow{i,eql}{}^{*} \circ \xleftarrow{no,a}$

Validierung der (*eql*)-Diagrammsätze

```
$ jonah -H80M validate --ruleset=eql --types=List --addrules --verbose
  --stats=10000 --class=1,1,2 --letrec
```

erzeugt die Ausgabe:

```
--rulesets=eql
-- 614,915 expressions in class (1, 1, 2, 0, 0).
-- 614,915 expressions tested.
-- rule set   |      tested expressions |  not found
-----------------------------------------------------
-- eql        |     44,682 |      7.2 % |          0
```

7.4.3 Reduktionsdiagramme für (*ldels*)

Wir wollen einen (*ldel*)-Redex statt in Reduktionskontexten auch in Oberflächenkontexten suchen. Dazu definieren wir einen neuen Reduktionstyp (*ldels*).

In der allgemeinen Markierungsfunktion `markRedex'` reicht eine neue Zeile:

```
LDel     v -> markContextR markLDel
LDelS    v -> markContextS markLDelS
```

Markierung von (*ldels*)

Die eigentliche Markierungsfunktion `markLDelS` ist eine Kopie von `markNLDel`, jedoch mit der neuen Markierung `LDelS` (siehe Kapitel 7.3.1, Seite 113).

Reduktion von (*ldels*)

Zur Reduktion verwandeln wir den (*ldels*)-Redex in einen (*ldel*)-Redex und reduzieren diesen.

```
reduceCoreExpr rec level (ELet (Redex (LDelS v)) r bs e)
    = reduceCoreExpr rec level (ELet (Redex (LDel v)) r bs e)
```

Diagrammsätze

Wir geben je einen kompletten Satz Vertauschungs- und Gabeldiagramme an, die für den Beweis der kontextuellen Äquivalenz der (*ldel*)-Reduktion benötigt werden.

Lemma 7.4.4 Ein kompletter Satz Vertauschungsdiagramme für die (*ldel*)-Reduktion enthält folgende Diagramme:

- $(i, ldels) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, ldels)$
- $(i, ldels) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
- $(i, ldels) \quad \rightsquigarrow \quad (no, \{lapp, llet, lcase\}^+) \circ (i, ldels)$
- $(i, ldels) \circ (no, \{lapp, llet, lcase\}^+) \quad \rightsquigarrow \quad (no, \{lapp, llet, lcase\}^*) \circ (i, ldels)$
- $(i, ldels) \circ (no, cp) \quad \rightsquigarrow \quad (no, abs) \circ (no, llet) \circ (no, cp) \circ (i, ldel) \circ (i, lcv)^* \circ (i, ldel)$
- $(i, ldels) \circ (no, case)$
  $\rightsquigarrow \quad (no, abs) \circ (no, lcase) \circ (no, llet) \circ (no, case) \circ (i, ldel) \circ (i, lcv)^* \circ (i, ldel)$

Validierung der (*ldels*)-Diagrammsätze

Die Validierung der Reduktionsdiagramme liefert allerdings schon bei kleinen Termklassen Fehler.

```
$ jonah -H80M validate --ruleset=ldels --types=List --addrules --verbose
  --stats=10000 --class=1,1,2 --letrec
```

erzeugt die Ausgabe:

```
--rulesets=ldels
-- 614,915 expressions in class (1, 1, 2, 0, 0).

-- ldels: no rule for `(letrec a=(Cons (letrec b=a in a)) in a)`
-- ERROR: (letrec a=(Cons ≪ldels/b:(letrec b=a in a)≫) in a)
      --> (letrec a=(Cons a) in a)
        : No rule found!

-- ldels: no rule for `(letrec a=(Cons (letrec b=Nil in a)) in a)`
-- ERROR: (letrec a=(Cons ≪ldels/b:(letrec b=Nil in a)≫) in a)
      --> (letrec a=(Cons a) in a)
        : No rule found!
[...]
-- 614,915 expressions tested.
-- rule set  |      tested expressions |  not found
----------------------------------------------------
-- ldels     |    419,068 |     68.1 % |        111
```

Die Fehler betreffen, wie schon bei der (*ldup*)-Reduktion, Terme in Konstruktoranwendungen.

7.5 Reduktionsdiagramme im $\lambda_{nd,rec,case}$-Kalkül

Der Kalkül $\lambda_{nd,rec,eql}$ aus Kapitel 7.4 soll nun durch eine neue (*case*)-Reduktion, die ebenfalls über Variablen-Indirektionen hinweg reduziert, erweitert werden. Durch diese Änderung wird wahrscheinlich die (*eql*)-Reduktion nicht mehr benötigt.

Wir ändern also die Markierungs- und Reduktionsmechanismen, so daß sie den Kalkül λ_{ndlr}[9] aus [SSH00] implementieren.

Bei einem `case`-Ausdruck muß nicht direkt die Konstruktoranwendung verglichen werden, sondern ebenfalls nur das Ende einer Variablen-Indirektionenfolge, wie beispielsweise im Term (`letrec` x `= Nil,` y `=` x `in` (`case` y `of -> True, Cons a as -> False`)).

7.5.1 Verfolgen von Indirektionen in (*case*)

Die Erweiterung der (*case*)-Reduktion bzgl. der Definitionen 3.1.10 und 7.4.1 hat weiterhin die Konsequenz, daß die (*abs*)-Reduktion nicht mehr benötigt wird, da die Konstruktorargumente während der (*case*)-Reduktion abstrahiert werden.

Definition 7.5.1 Der *nichtdeterministische Kalkül* $\lambda_{nd,rec,case}$ enthält die Reduktionsregeln aus Definition 7.4.1 (Seite 122), mit Ausnahme der (*abs*)-Reduktion. Die Reduktionsregel für (*case*) wird folgendermaßen erweitert, wobei die erste Reduktionsregel der Vollständigkeit halber noch einmal aufgeführt ist.

[9] Wir benennen die Kopierreduktion nicht um, und schreiben weiterhin (*cp*) statt (*cpn*).

$$C[\ \texttt{case}_A\ (c_{A,i}\ t_1 ... t_n)\ \texttt{of}\ ..., c_{A,j}\ \vec{y_j}\,.\,t_j\,,...\ ]$$
$$\xrightarrow{case}\quad C[\ \texttt{letrec}\ y_{j,1}\ \texttt{=}\ t_1\,,...,y_{j,n}\ \texttt{=}\ t_n\ \texttt{in}\ t_j\ ]$$

$$C[\ \texttt{letrec}\ x_1\ \texttt{=}\ (c_{A,j}\ t_1 ... t_n), x_2\ \texttt{=}\ x_1\,,...,x_i\ \texttt{=}\ x_{i-1}, x_{i+1}\ \texttt{=}\ s_{i+1}\,,...$$
$$\quad \texttt{in}\ D[\ \texttt{case}_A\ x_i\ \texttt{of}\ ..., c_{A,j}\ \vec{y_j}\,.\,t_j\,,...\ ]\ ]$$
$$\xrightarrow{case}\quad C[\ \texttt{letrec}\ z_1\ \texttt{=}\ t_1\,,...,z_n\ \texttt{=}\ t_n\,, x_1\ \texttt{=}\ (c_{A,i}\ z_1 ... z_n), x_2\ \texttt{=}\ x_1\,,...,x_i\ \texttt{=}\ x_{i-1}, x_{i+1}\ \texttt{=}\ s_{i+1}\,,...$$
$$\quad \texttt{in}\ D[\ \texttt{letrec}\ y_{j,1}\ \texttt{=}\ z_1\,,...,y_{j,n}\ \texttt{=}\ z_n\ \texttt{in}\ t_j\ ]\ ]$$

$$C[\ \texttt{letrec}\ x_1\ \texttt{=}\ (c_{A,j}\ t_1 ... t_n), x_2\ \texttt{=}\ x_1\,,...,x_i\ \texttt{=}\ x_{i-1}\,,...$$
$$\quad x_{i+1}\ \texttt{=}\ D[\ (\texttt{case}_A\ x_i\ \texttt{of}\ ..., c_{A,j}\ \vec{y_j}\,.\,t_j\,,...)\ ],...\ \texttt{in}\ r$$
$$\xrightarrow{case}\quad C[\ \texttt{letrec}\ z_1\ \texttt{=}\ t_1\,,...,z_n\ \texttt{=}\ t_n\,, x_1\ \texttt{=}\ (c_{A,i}\ z_1 ... z_n), x_2\ \texttt{=}\ x_1\,,...,x_i\ \texttt{=}\ x_{i-1}\,,$$
$$\quad x_{i+1}\ \texttt{=}\ D[\ \texttt{letrec}\ y_{j,1}\ \texttt{=}\ z_1\,,...,y_{j,n}\ \texttt{=}\ z_n\ \texttt{in}\ t_j\ ]\ ],...\ \texttt{in}\ r$$

Die Markierung des NO-Redex geht genauso vonstatten wie in Definition 3.3.2 auf Seite 36, jedoch ohne (*abs*) und mit geänderter (*cp*)- und (*case*)-Markierung. Anpassungen sind auch noch bei der Erkennung einer CWHNF und von irreduziblen Ausdrücken zu machen. Aus diesem Grund wollen wir den kompletten Markierungsalgorithmus angeben.

Definition 7.5.2 (MarkNO$_{rec,case}$) Der *NO-Redex* eines $\Lambda_{nd,rec}$-Ausdrucks t wird mittels einer Menge von Regeln definiert, die die Markierungen E (*evaluation*) oder H (*halt*) bis zu einem finalen Teilausdruck in t auf- und abbewegen. Der Algorithmus beginnt mit dem unmarkierten Ausdruck t^E:

1. Erkennung eines NO-Redex:

(*ndl/r*)	$R[\ \texttt{choice}\ s\ t\ ]$
(*lbeta*)	$R[\ (\backslash x.s)\ t\ ]$
(*lapp*)	$R[\ (\texttt{letrec}\ \ldots)\ t\ ]$
(*cp*)	$R[\ \texttt{letrec}\ x_1\ \texttt{=}\ s_1, x_2\ \texttt{=}\ x_1,...,x_i\ \texttt{=}\ x_{i-1}, x_{i+1}\ \texttt{=}\ s_{i+1},...\ \texttt{in}\ D[\ x_i\ ]\ ]$ falls $s_i = (\backslash v\,.\,t)$ oder $s_i = (c_{A,i}\ t_1 \ldots t_n)$
(*cp*)	$R[\ \texttt{letrec}\ x_1\ \texttt{=}\ s_1, x_2\ \texttt{=}\ x_1,...,x_i\ \texttt{=}\ x_{i-1}, x_{i+1}\ \texttt{=}\ D[x_i],...\ \texttt{in}\ s\ ]$ falls $s_i = (\backslash v\,.\,t)$ oder $s_i = (c_{A,i}\ t_1 \ldots t_n)$
(*lletr1*)	$R[\ \texttt{letrec}\ \ldots\ \texttt{in}\ (\texttt{letrec}\ \ldots\ \texttt{in}\ t)\ ]$
(*lletr2*)	$R[\ \texttt{letrec}\ x_1\ \texttt{=}\ (\texttt{letrec}\ \ldots), x_2\ \texttt{=}\ R_2[x_1], \ldots, x_j\ \texttt{=}\ R_j[x_{j-1}], \ldots\ \texttt{in}\ R_\infty[x_j]\ ]$
(*lcase*)	$R[\ \texttt{case}_A\ (\texttt{letrec}\ \ldots)\ \texttt{of}\ alts\ ]$
(*case*)	$R[\ \texttt{case}_A\ (c_{A,i}\ v_1 \ldots v_n)\ \texttt{of}\ alts\ ]$
(*case*)	$R[\ \texttt{letrec}\ x_1\ \texttt{=}\ (c_{A,j}\ t_1 ... t_n), x_2\ \texttt{=}\ x_1,...,x_i\ \texttt{=}\ x_{i-1}, x_{i+1}\ \texttt{=}\ s_{i+1},...$ $\texttt{in}\ D[\ \texttt{case}_A\ x_i\ \texttt{of}\ alts\ ]\ ]$
(*case*)	$R[\ \texttt{letrec}\ x_1\ \texttt{=}\ (c_{A,j}\ t_1 ... t_n), x_2\ \texttt{=}\ x_1,...,$ $x_i\ \texttt{=}\ x_{i-1}, x_{i+1}\ \texttt{=}\ D[\ \texttt{case}_A\ x_i\ \texttt{of}\ alts\ ],...\texttt{in}\ s\ ]$

2. Erkennung einer WHNF:

   (a) $C[\ (\lambda x\,.\,r)^E\ ]$. Markiere s als WHNF.

   (b) $C[\ (c\ t_1 \ldots t_n)^E\ ]$ und $n\ \leq\ ar(c)$. Markiere als CWHNF.

   (c) $C[\ x^E\ ]$ und x ist eine *freie Variable*. Markiere als VWHNF.[10]

[10]Freie Variablen, die Argumente eines Konstruktors sind, werden nicht als Kopierziel markiert.

3. Erkennung eines irreduziblen Ausdrucks:

   (a) $C[\ (c\ s_1 \ldots s_n)^H\ ]$, wenn $n\ >\ ar(c)$.

   (b) $C[\ (\mathtt{case}_A\ s\ \mathtt{of}\ t_1\mathtt{,}\ \ldots\ \mathtt{,}t_n)^H\ ]$, wenn $n\ <\ |A|$

   (c) $C[\ (\mathtt{case}_A\ s\ \mathtt{of}\ t_1\mathtt{,}\ \ldots\ \mathtt{,}t_{|A|})^H\ ]$, falls s entweder der Form
      - $\lambda x\mathtt{.}r$ oder
      - $(c_B\ r_1 \ldots r_n)$ oder
      - $(c_A\ r_1 \ldots r_n)$ mit $n\ \neq\ ar(c)$ entspricht.

   (d) $C[\ (\mathtt{letrec}\ x_1\ \mathtt{=}\ s_1\mathtt{,}x_2\ \mathtt{=}\ x_1\mathtt{,}\ldots\mathtt{,}x_i\ \mathtt{=}\ x_{i-1}\mathtt{,}x_{i+1}\ \mathtt{=}\ s_{i+1}\mathtt{,}\ldots$
   $\mathtt{in}\ (\mathtt{case}_A\ x_i\ \mathtt{of}\ t_1\mathtt{,}\ \ldots\ \mathtt{,}t_{|A|}))^H\ ]$, falls s_1 entweder der Form
      - $\lambda x\mathtt{.}r$ oder
      - $(c_B\ r_1 \ldots r_n)$ oder
      - $(c_A\ r_1 \ldots r_n)$ mit $n\ \neq\ ar(c)$ entspricht.

4. Fortsetzung des Markierungsverfahrens, falls keine der obigen Regeln angewandt werden kann:

   (a) $R[\ (s\ t)^E\ ]$. Markiere weiter in $C[\ (s^E\ t)^e\ ]$.

   (b) $R[\ (\mathtt{letrec}\ \ldots\ \mathtt{in}\ t)^E\ ]$. Markiere weiter in $R[\ (\mathtt{let}\ \ldots\ \mathtt{in}\ t^E)^e\ ]$.

   (c) $R[\ \mathtt{letrec}\ x_1\ \mathtt{=}\ s_1\mathtt{,}x_2\ \mathtt{=}\ R_2[x_1]\mathtt{,}\ \ldots\ \mathtt{,}x_j\ \mathtt{=}\ R_j[x_{j-1}]\mathtt{,}\ \ldots\ \mathtt{in}\ R_\infty[x_j^E]\ ]\ ]$ und x ist eine VWHNF. Markiere weiter in
   $R[\ \mathtt{letrec}\ x_1\ \mathtt{=}\ s_1^E\mathtt{,}x_2\ \mathtt{=}\ R_2[x_1]\mathtt{,}\ \ldots\ \mathtt{,}x_j\ \mathtt{=}\ R_j[x_{j-1}]\mathtt{,}\ \ldots\ \mathtt{in}\ R_\infty[x_j^e]\ ]\ ]$.

   (d) $C[\ (\mathtt{case}_A\ s\ \mathtt{of}\ t_1\mathtt{,}\ \ldots\ \mathtt{,}t_{|A|})^E\ ]$. Markiere weiter in $R[\ (\mathtt{case}_A\ s^E\ \mathtt{of}\ t_1\mathtt{,}\ \ldots\ \mathtt{,}t_{|A|})^e\ ]$.

   Das Markierungsverfahren stoppt, falls entweder

   (a) ein NO-Redex oder

   (b) ein irreduzibler Ausdruck oder

   (c) eine WHNF, CWHNF oder VWHNF

   entdeckt wird.

Markierung von (*cp*) und (*case*)

Da im Kalkül $\lambda_{nd,rec,case}$ die Argumente von Konstruktoranwendungen erst bei einem Vergleich (also `case`) abstrahiert werden, hängt die Kopierbarkeit nicht mehr davon ab, ob die Konstruktorargumente alles Variablen sind (vergleiche Seite 38). Wir markieren im rekursiven Fall eine Konstruktoranwendung immer als CWHNF.

```
markNORedex rec expr@(EApp m e1 e2)
    = case getTagArityArgs expr
        of Just (t,a,as)
             -> if a < length_as
                   then HasError ("Reduction.markNORedex (App): "
                                  ++ "oversaturated constructor ‘"
                                  ++ show expr ++ "‘") expr
                   else if all isVariable as || rec
                           then HasCWHNF (t,a) (a - length_as) expr
                           else HasRedex Abs (EApp (Redex Abs) e1 e2)
                   where
                     length_as = length as
           Nothing
             -> case markNORedex rec e1
                  of HasError    s    e1’ -> HasError s expr
                     HasRedex    r    e1’ -> HasRedex r (EApp m e1’ e2)
                     HasVWHNF    v    e1’ -> HasVWHNF v (EApp m e1’ e2)
```

Für die Erkennung eines (*case*)-Redex muß nur die Markierung eines `letrec`-Ausdrucks geändert werden, da hier auch die „neuen" (*case*)-Redexe gesucht werden. Wir ändern also die auf Seite 123 aufgeführte Markierungsfunktion für `letrec`-Ausdrücke, geben allerdings aus Platzgründen nur die geänderten Passagen an.

```
markNORedex True expr@(ELet m True bs e)
    = case markNORedex True e
        of HasError s    e’ -> HasError s  expr
           HasRedex r    e’ -> HasRedex r  (ELet m    True bs e’)
           HasWHNF       e’ -> HasWHNF     (ELet WHNF True bs e )
           HasCWHNF ta a e’ -> HasCWHNF ta a expr
           HasVWHNF v    e’ -> if v ‘notElem‘ boundVars
                                  then HasVWHNF v (ELet m True bs e’)
                                  else checkBindings v v "" [v] e’
        where
[...]
```

Bei der Überprüfung eines Bindungsterms wird im Falle einer CWHNF[11] mittels der Funktion `checkCase` überprüft, ob sich die als VWHNF markierte Variable `v` (siehe oben) in einem `case`-Ausdruck befindet.

```
        checkBoundExprs
          = case markNORedex True vexpr
              of HasError s  ve -> HasError s  e
                 HasRedex r’ ve -> hasRedex r’ m (newbs refVar ve) e
                 HasCWHNF _ _ _ -> checkCase
                 HasVWHNF v’ ve -> checkCycles v’ ve
```

Falls die als VWHNF markierte Variable `v` sich in einem `case`-Ausdruck befindet, muß nachgeprüft werden, ob die Konstruktoranwendung überhaupt zu dem `case`-Ausdruck paßt. Dazu

[11]Die CWHNF kann sich nicht in einem Kontext befinden, da dieser ein Redex sein müßte. Wir können also davon ausgehen, daß die Konstruktoranwendung direkt an die entsprechende Variable gebunden ist.

suchen wir den `case`-Ausdruck und setzen die Konstruktoranwendung direkt ein. Zur Überprüfung der Stelligkeit und des Typs der Konstruktoranwendung verwenden wir die „normale“ Markierungsfunktion für `case`-Ausdrücke. Dabei gibt es nur die zwei Möglichkeiten[12], daß ein Fehler auftritt oder der Ausdruck ein (*case*)-Redex ist.

```
checkCase
  = case getMarkedExprs isVCase e'
      of []  -> copyRedex
         c:_ -> markVCase c
      where
        isVCase (ECase _ _ (EVar VWHNF _) _) = True
        isVCase _                            = False

        markVCase (ECase m t (EVar VWHNF _) as)
          = case markNORedex True (ECase m t vexpr as)
              of HasError err  _ -> HasError err expr
                 HasRedex Case _ -> caseRedex
                 otherwise       -> error $ "Reduction.markNORedex: "
                                           ++ show expr
```

Reduktion von (*case*)

Wir verwenden zur Implementierung der (*case*)-Reduktion die Original-(*case*)-Reduktion (ohne Variablen-Indirektion), indem die Konstruktoranwendung, die über mehrere Indirektionen hinweg gesucht werden muß (`getConstr`), derart umgewandelt wird, daß alle Argumente durch neue Variablen abstrahiert werden. Wir erhalten eine pure Konstruktoranwendung, die wir jetzt an Stelle der Variable `caseVar` in den Term einfügen. Da sich der `case`-Ausdruck sowohl im Term `e` als auch in den gebundenen Termen befinden kann, müssen alle Teilterme betrachtet werden. Der umgewandelte Gesamtausdruck ist dann ein „normaler“ (*case*)-Redex, der mit der Original-Funktion reduziert werden kann.

```
reduceCoreExpr True level (ELet (Redex Case) _ bs e)
    = reduceCoreExpr True level (ELet Unmarked True bs' e')
      where
```

Wir müssen in `e` und den gebundenen Termen nach der als VWHNF markierten Variable suchen.

```
caseVar
  = case concatMap (getMarkedExprs isVWHNF) (e:[x | (_,x) <- bs])
      of [EVar _ v] -> v
         otherwise  -> error $ "Reduktion.reduceCoreExpr.(case): "
                             ++ "exactly one VWHNF allowed"
```

Dann wird die Konstruktoranwendung in ihre Bestandteile zerlegt und für jedes Argument eine neue Variable erzeugt.

[12]Zur Sicherheit wird eine Fehlermeldung ausgegeben, falls es doch eine dritte Möglichkeit geben sollte.

```
(cv,(t,a,as)) = getConstr caseVar

newVars = [ newVarPrefix ++ show level ++ "_" ++ show x
          | x <- take (length as) [1..]
          ]
```

Die Funktion `getConstr` sucht die Konstruktoranwendung in den Bindungen und liefert ihre Merkmale (Typ, Stelligkeit und Argumente) sowie die Variable, an die sie gebunden ist.

```
getConstr v
  = case lookupBinding v bs
      of (EVar _ v') -> getConstr v'
         c           -> case getTagArityArgs c
                          of Just x  -> (v,x)
                             Nothing -> cError

cError
  = error $ "Reduktion.reduceCoreExpr.(case): "
            ++ "no constructor application"
```

Die pure Konstruktoranwendung wird mit den neuen Variablen erzeugt und mittels der Funktion (`applyFunc newCase`) in die Sub-Terme[13] an Stelle der VWHNF eingefügt. Die Funktion `applyFunc` wendet ihr erstes Argument — eine Funktion, die Ausdrücke verändert — auf alle Sub-Terme eines Ausdrucks an.

```
newConstructor
  = makeConstructor (t,a,[EVar Unmarked v | v <- newVars])

e'      = applyFunc newCase e
```

Die abstrahierten Konstruktorargumente werden zu den Bindungen hinzugefügt.

```
bs'     = zip newVars as
          ++ [ (v,e')
             | (v,e) <- bs
             , let e' = if v == cv
                          then newConstructor
                          else applyFunc newCase e
             ]
```

Das Einfügen der puren Konstruktoranwendung wird mittels der Kombination `applyFunc` und der Funktion `newCase` erreicht.

```
newCase (ECase m            t (EVar VWHNF v) as)
      = (ECase (Redex Case) t newConstructor as)
newCase x = x
```

[13] An dieser Stelle kann noch optimiert werden, indem der die VWHNF enthaltende Teilterm bei der Suche vermerkt wird, und nun gezielt geändert wird.

Diagrammsätze

Wir geben je einen Satz Vertauschungs- und Gabeldiagramme an:

Lemma 7.5.1 Ein kompletter Satz Vertauschungsdiagramme für die (cpd)- und (cpt)-Reduktion enthält folgende Diagramme:

- $(i, cpt) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (\{i, no\}, cpt)$
- $(i, cpt) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, cpd) \circ (no, a) \quad \rightsquigarrow \quad (no, a) \circ (i, cpd)$
- $(i, cpd) \circ (no, cp) \quad \rightsquigarrow \quad (no, cp) \circ (i, cpd) \circ (i, cpd)$
- $(i, cpd) \circ (no, a) \quad \rightsquigarrow \quad (no, a)$
  mit $a \in \{case, ndl, ndr\}$
- $(i, cpd) \circ (no, lbeta) \quad \rightsquigarrow \quad (no, lbeta) \circ (\{i, no\}, cpt)$

Lemma 7.5.2 Ein kompletter Satz Gabeldiagramme für die (cpd)- und (cpt)-Reduktion enthält folgende Diagramme:

- $\xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad \rightsquigarrow \quad \xrightarrow{i,cpt} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cpt} \circ \xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,cpd} \quad \rightsquigarrow \quad \xrightarrow{i,cpd} \circ \xleftarrow{no,a}$
- $\xleftarrow{no,cp} \circ \xrightarrow{i,cpd} \quad \rightsquigarrow \quad \xrightarrow{i,cpd} \circ \xrightarrow{no,cpd} \circ \xleftarrow{no,cp}$
- $\xleftarrow{no,cpt} \circ \xleftarrow{no,lbeta} \circ \xrightarrow{i,cpd} \quad \rightsquigarrow \quad \xleftarrow{no,lbeta}$
- $\xleftarrow{no,a} \circ \xrightarrow{i,cpt} \quad \rightsquigarrow \quad \xleftarrow{no,a}$
  mit $a \in \{case, ndl, ndr\}$
- $\xleftarrow{no,lbeta} \circ \xrightarrow{i,cpd} \quad \rightsquigarrow \quad \xrightarrow{i,cpt} \circ \xleftarrow{no,lbeta}$

Validierung der (cp)-Diagrammsätze

```
$ jonah -RTS validate --ruleset=cpn --types=List --time
  --verbose --addrules --class=0,1,2
```

erzeugt die Ausgabe:

```
--rulesets=cpn
-- 68,447 expressions in class (0, 1, 2, 0, 0).
-- cpn: (letrec a=b, b=Nil in b)
> i.cpt n.cp ==> n.cp
-- cpn: (letrec a=Nil, b=(Cons a) in b)
> i.cpt n.cp ==> n.cp i.cpt i.cpt
-- 68,447 expressions tested.
-- rule set   |      tested expressions |  not found
----------------------------------------------------
-- cpn        |     36,137 |     52.7 % |          2
```

Die Diagrammsätze aus den vorherigen Kapiteln sind anscheinend doch nicht vollständig. Zumindest hat JONAH noch zwei neue Diagramme gefunden.

Leider können wir die Diagramme nicht an großen Termmengen testen, da die Zeit dazu im Rahmen dieser Arbeit nicht mehr ausgereichte.

Tests an kleinen Mengen zeigen jedoch, daß durch den neuen Ansatz möglicherweise vollständige Diagrammsätze gefunden werden können.

Kapitel 8

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, ein System zu entwickeln, mit dem Sätze von Reduktionsdiagrammen auf ihre Vollständigkeit überprüft werden können. Das System sollte in der puren, funktionalen Programmiersprache *Haskell* realisiert werden und speziell auf einen nichtdeterministischen Kalkül mit Konstruktoren, einem `case`-Konstrukt und einem rekursiven `let`-Ausdruck (`letrec`) ausgerichtet sein. Dazu wurde in *Kapitel 2* der λ-Kalkül und funktionale Programmiersprachen eingeführt, um die Entwicklung der Kalküle λ_{nd} und $\lambda_{nd,rec}$ zu motivieren.

Die Basis-Kalküle λ_{nd} und $\lambda_{nd,rec}$ wurden in *Kapitel 3* bezüglich ihrer Sprachdefinition und Reduktionsregeln vorgestellt. Die Implementierung der beiden Kalküle und der *Parser* für die verwendete Kernsprache wurden ebenso angegeben wie die Mechanismen zur Markierung und Reduktion der Terme. Die Unterscheidung eines reduzierbaren Terms in NO- und internen Redex, sowie deren Erkennung wurde in diesem Kapitel besonders herausgestellt, da sie die Grundlage für die zu validierenden Reduktionsdiagramme darstellt. Damit ist der Grundstock für diese Arbeit gelegt worden.

In *Kapitel 4* haben wir kurz Methoden vorgestellt, die λ-Kalkülen und deren Derivate eine Bedeutung zuzumessen versuchen, um Programme oder Terme miteinander vergleichen zu können. Also nicht nur ihre Termstruktur, sondern ihr Verhalten zu vergleichen. Dabei ist festgestellt worden, daß für nichtdeterministische Kalküle nur eine Methode verfügbar ist, und zwar die der *kontextuellen Äquivalenz*. Da wir gängige Programmtransformationen dahingehend überprüfen wollten, ob sie der kontextuellen Äquivalenz genügen, also nicht die Auswertung oder das Ergebnis derselben verändern, haben wir ein Kontext-Lemma vorgestellt. Mit Hilfe des Kontext-Lemmas ist es uns möglich gewesen, die Gültigkeit von Beweisen in Reduktionskontexten auf beliebige Kontexte auszudehnen. Für die eigentlichen Beweise werden Reduktionsdiagramme benötigt, die Reduktionsfolgen in verschiedener Weise umstellen. Wir haben die Reduktionsdiagramme ebenfalls in *Kapitel 4* definiert.

Die Reduktionsdiagramme müssen an jeder Stelle einer Reduktionsfolge anwendbar sein, da sonst die Äquivalenz-Beweise nicht möglich sind. D. h. es müßte bewiesen werden, daß mindestens ein Reduktionsdiagramm einer definierten Menge von Reduktionsdiagrammen für jeden Term und jede seiner Reduktionsfolgen anwendbar ist. Der naive Weg wäre, alle Terme und seine Reduktionsfolgen dahingehend zu überprüfen. Da dies jedoch nicht möglich ist, haben wir uns dazu entschlossen so viele Terme wie möglich zu untersuchen, nicht um die Vollständigkeit der Diagramme zu beweisen, sondern um zu zeigen, daß auf viele verschiedenartige Terme Reduktionsdiagramme erfolgreich angewendet werden können.

Die Generierung der Terme haben wir in *Kapitel 5* ausführlich dargestellt. In *Kapitel 6* haben wir die Sprache zur Darstellung von Reduktionsdiagrammen und die Implementierung der Diagramm-Validierung beschrieben. Weiterhin haben wir ein Verfahren vorgestellt, mit dem wir fehlende Diagramme für Terme suchen und meist auch finden können.

Das gesamte System haben wir so aufgebaut, daß eine große Menge an Termen generiert wird, für die Diagramme anwendbar sein müssen. Wenn keines der Diagramme auf einen Term anwendbar ist, also nicht validiert werden kann, dann wird ein passendes Diagramm gesucht. Mit Hilfe des entwickelten Systems JONAH sind in *Kapitel 7* verschiedene Programmtransformationen dahingehend überprüft worden, ob die angegebenen Reduktionsdiagramme für große Mengen von Termen gültig sind. Dabei ist als Ergebnis herausgekommen, daß ein vollständiger Satz an Reduktionsdiagrammen für die (cp)-Reduktion nur sehr schwer zu finden ist. Selbst neu hinzugefügte Transformationen, die dabei helfen sollten, die Schwierigkeiten zu umgehen, konnten nicht verwendet werden, da es uns ebenfalls nicht möglich gewesen ist, für sie vollständige Diagrammsätze zu finden. Daraufhin haben wir den Basis-Kalkül zweimal erweitert, um die Schwierigkeiten der (cp)-Diagramme in den Griff zu bekommen. Aber auch diese Ansätze haben kein zufriedenstellendes Ergebnis geliefert.

Als positives Ergebnis dieser Arbeit kann aber festgestellt werden, daß das System JONAH bei der Entwicklung nichtdeterministischer Kalküle auf Basis von λ-Kalkülen eine sehr gute Hilfestellung sein kann, auch wenn es nicht zum vollständigen Beweis der Gültigkeit von Reduktionsdiagrammen dienen kann.

8.1 Ausblick

Zum Schluß wollen wir noch Verbesserungen für das System vorschlagen, die in der Kürze der Zeit nicht realisiert werden konnten.

Termgenerierung

Um die Menge der generierten Term möglichst klein zu halten, könnte der Termgenerator dahingehend geändert werden, daß Typ-Informationen durch die gesamte Termstruktur weitergereicht werden können. Die Beschränkung auf eine Rekursionsstufe kann die Menge nicht zufriedenstellend einschränken.

Validierung

Die Verwendung von allgemeinen Diagramm-Definitionen schlägt sich auf das Laufzeitverhalten der Validierung nieder, da der Diagrammsatz mit gleich strukturierten Diagrammen erweitert wird, und somit die Überprüfung der Diagramme länger dauert. Die Behandlung des Mengenkonstrukts sollte auch für die „synchronen“ Diagramme möglich sein. Um die Validierung der Reduktionsdiagramme weiter zu verallgemeinern, sollten die Reduktionstypen mit einer Quantifizierungsmarkierung versehen werden können, um auch andere Arten von Diagrammen zu ermöglichen.

Diagrammsuche

Die Diagrammsuche sollte gezielter vonstatten gehen, da sehr viele unnützen Diagramme getestet werden und somit das Laufzeitverhalten stören.

Reduktionsmechanismus

Die Trennung der beiden Markierungsverfahren ist etwas unglücklich, da bei der Markierung schon gezielt Informationen „nach oben“ geholt werden können, so daß bei der folgenden Reduktion der Term nicht mehrmals durchlaufen werden muß.

Analyse bei Fehlern

Bei den aufgetretenen Schwierigkeiten bzgl. der (*ldup*)-Reduktion wäre es von Vorteil gewesen, daß sich die interaktive Reduktion von Jonah nicht auf einen Reduktionspfad beschränkt hätte, sondern auf übersichtliche Weise alle Reduktionspfade dargestellt hätte. So wäre es vielleicht möglich gewesen, die Probleme schneller zu analysieren.

Beweis von Reduktionsdiagrammen

Weitere Forschungen sollten einen Beweiser für Reduktionsdiagramme möglich machen, der nicht nur einen speziellen Kalkül analysieren kann, sondern wie Jonah dem Entwickler die Möglichkeit gibt, das System mit wenigen Änderungen anzupassen. Jedenfalls könnte Jonah bei der Entwicklung eines solchen Werkzeugs gute Dienste leisten.

Ein Beweiser muß jedoch wieder den Unterschied zwischen Vertauschungs- und Gabeldiagramm machen, da er die entsprechenden Quantifizierungen der Reduktionstypen beachten muß.

Anhang A

Programmauszüge

Nachfolgend sollen noch einige wichtige Funktionen der Implementierung vorgestellt werden, die im Hauptteil dieser Arbeit nur erwähnt werden. Aus Platzgründen kann jedoch nur eine Auswahl dieser Funktionen dargestellt werden, oft auch nur gekürzt. JONAH besteht aus mehreren Modulen, die in der folgenden Tabelle kurz beschrieben werden.

Modul	Datei	Beschreibung
Main	Main.hs	Bearbeiten der Konsolen-Eingabe und starten der entsprechenden Programmteile, sowie CGI-Funktionalität. Da System-Funktionen bei den verwendeten Compiler unterschiedlich deklariert sind, wird aus dieser Datei eine Compiler-spezifische Datei erstellt, z. B. *MainHbc.hs*.
MainAll	MainAll.hs	Compiler-unabhängige Schnittstelle (interaktiver Modus von JONAH).
ExpressionParser	lib/ExpressionParser.y	Definition der Kernsprache, wird mittels *lucky* in ein Haskell98-Modul übersetzt.
Expression	lib/Expression.hs	Konstruktoren für die Kernsprache, Reduktionstypen, sowie Hilfsfunktion für ebendiese.
RuleParser	lib/RuleParser.hs	Definition der Diagramm- bzw. Regel-Sprache, wird mittels *lucky* in ein Haskell98-Modul übersetzt.
Rule	lib/Rule.y	Konstruktoren für Diagramme bzw. Regeln und Regel-Generator.
Reduction	lib/Reduction.hs	Markierungs- und Reduktionsmechanismus, sowie Generierung von Reduktionsfolgen und „interaktives Reduzieren“.

Modul	Datei	Beschreibung
Validation	lib/Validation.hs	Termgenerierung, Zählfunktion, Diagramm-Validierung und Diagrammsuche.
CGI	cgi/*.hs	CGI-Implementierung von Erik Meijer [MvD96]. Anpassung der `<TEXTAREA>`-Umgebung.
Console	lib/Console.hs	*Escape*-Sequenzen für die Linux-Konsole.
Utils	lib/Utils.hs	Hilfsfunktionen wie Kreuzprodukt, Diagonalisierung, Permutationen etc.

A.1 Modul *Expression*

Die Gleichheit für Reduktionstypen wird durch eine eigene Instanz der `Eq`-Klasse realisiert, da wir bestimmte Reduktionstypen als gleich erachten, wobei auch die Spezialisierung berücksichtigt wird.

```
instance Eq ReductionType
    where
      (==) (VAR      a)  (VAR      b) = a == b
      (==) (LBeta     )  (LBeta     ) = True
      (==) (LApp      )  (LApp      ) = True
      (==) (LLet      )  (LLet      ) = True
      (==) (LLetrec1  )  (LLetrec1  ) = True
      (==) (LLetrec1  )  (LLet      ) = True
      (==) (LLetrec2 a)  (LLetrec2 b) = a == b
      (==) (LLetrec2 a)  (LLet      ) = True
      ...
      (==) x             y            = False
```

Zum allgemeinen Vergleichen der Reduktionstypen wird die Funktion `isSameReductionType` verwendet.

```
isSameReductionType (VAR      a)  (VAR      b) = True
isSameReductionType (LBeta     )  (LBeta     ) = True
isSameReductionType (LApp      )  (LApp      ) = True
isSameReductionType (LLet      )  (LLet      ) = True
isSameReductionType (LLetrec1  )  (LLetrec1  ) = True
isSameReductionType (LLetrec1  )  (LLet      ) = True
isSameReductionType (LLetrec2 a)  (LLetrec2 b) = True
isSameReductionType (LLetrec2 a)  (LLet      ) = True
...
isSameReductionType x             y            = False
```

`makeGeneralReductionType` macht aus speziellen Reduktionstypen allgemeine.

```
makeGeneralReductionType (LLetrec2 a) = (LLetrec2 "")
makeGeneralReductionType (Copy     a) = (Copy     "")
...
makeGeneralReductionType x            = x
```

Auch die Instanz der `Show`-Klasse muß neu definiert werden.

```
instance Show ReductionType
    where
      showsPrec _ = showString . toString
        where
          toString a
            = case a
                of VAR v      -> v
                   LBeta      -> "lbeta"
                   LApp       -> "lapp"
                   LLet       -> "llet"
                   LLetrec1   -> "lletr1"
                   LLetrec2 v -> "lletr2"
                   ...
```

Die Ausgabe der speziellen Reduktionstypen kann individuell definiert werden.

```
showReductionType
    = showreductiontype
        where
          s = "/"
          showreductiontype r@(VAR v)       = v
          showreductiontype r@(LLetrec2 "") = show r
          showreductiontype r@(LLetrec2  v) = show r ++ s ++ v
          ...
```

Funktionen zur Prüfung und Erkennung von Reduktionstypen.

```
isReductionType s
    = s `elem` [ show r
               | r <- reductionTypes ++ specReductionTypes]

decodeReductionType s
    = case [ r
           | r <- reductionTypes ++ specReductionTypes
           , show r == s
           ]
        of [] -> error $ "Expression.decodeReductionType: "
                         ++ "unknown reduction type '" ++ s ++ "'"
           x  -> head x
```

Alle Reduktionstypen sind durch `reductionTypes` erreichbar.

```
reductionTypes     = baseReductionTypes ++ extReductionTypes

baseReductionTypes = [ LBeta
                     , LApp
                     , LLet
                     , Abs
                     , Copy ""
                     , Case
                     , CaseLet
                     , ND
                     ]
```

```
specReductionTypes = [ LLetrec1
                     , LLetrec2 ""
                     , NDL
                     , NDR
                     ]

extReductionTypes  = [ CopyD    ""
                     , CopyT    ""
                     , LDel     ""
                     , LDelS    ""
                     , LDelCyc1 []
                     , LDelCyc2
                     , LDup     ""
                     , LCVT     ""
                     , LCVD     ""
                     , LCom
                     , UCP      ""
                     , EQL      ""
                     ]
```

Zum Vergleichen von Ausdrücken reicht es nicht, die Standard-Instanz von Haskell zu verwenden, da nicht auf α-Konvertibilität geachtet wird. Die Variablennamen in Abstraktionen und `let(rec)`-Ausdrücken werden in einer Liste paarweise mitgeführt, wobei die Funktion `checkAlpha` letztendlich die Variablen gemäß der Liste vergleicht.

```
instance Eq CoreExpr
    where
    (==) e e' = eq [] e e'
      where
        eq vs (EVar m v) (EVar m' v')
         = (m == m')
           && checkAlpha vs v v'

        eq vs (EChoice m e1 e2) (EChoice m' e1' e2')
         = (m == m')
           && (eq vs e1 e1')
           && (eq vs e2 e2')


        eq vs (ECons m n a) (ECons m' n' a')
         = (m == m')
           && (n == n')
           && (a == a')

        eq vs ec@(EComb m n ps e) ec'@(EComb m' n' ps' e')
         = eq vs (convertEComb2ELam ec) (convertEComb2ELam ec')

        eq vs (ELam m v e) (ELam m' v' e')
         = (m == m')
           && eq ((v, v'):vs) e e'

        eq vs (EApp m e1 e2) (EApp m' e1' e2')
         = (m == m')
           && (eq vs e1 e1')
           && (eq vs e2 e2')
```

Wegen der willkürlichen Reihenfolge der `let(rec)`-Bindungen, müssen alle möglichen Permutationen (mit `validBindings` erzeugt) verglichen werden.

```
eq vs (ELet m rec bs e) (ELet m' rec' bs' e')
 = (m == m')
   && (rec == rec')
   && (length bs == length bs')
   && or [eq vs' e e' | vs' <- validBindings vs bs bs']

eq vs (ECase m t e as) (ECase m' t' e' as')
 = (t == t')
   && (eq vs e e')
   && (length as == length as')
   && checkAlts vs as as'

eq _ _ _ = False

checkAlpha as v1 v2
  = case [a | a@(x,_) <- as, x == v1]
      of []        -> v1 == v2
         (_,v2'):_ -> v2 == v2'

validBindings as bs bs'
  = [ as''
    | combination <- map (zip bs) (permut bs')
    , let as'  = [(v1,v2) | ((v1,_),(v2,_)) <- combination]
    , let as'' = as' ++ as
    , and [eq as'' e1 e2 | ((_,e1),(_,e2)) <- combination]
    ]
```

Die Reihenfolge der Alternativen kann ebenfalls unterschiedlich sein.

```
checkAlts as alts alts'
  = let checkAlt ((n1,v1,e1),(n2,v2,e2))
          = (n1 == n2) && (eq ((zip v1 v2) ++ as) e1 e2)
     in all checkAlt (zip (sort alts) (sort alts'))
```

Zum Schluß sei noch eine Funktion erwähnt, die bei der Reduktion sehr oft Verwendung findet. `replaceMarkedVar` ersetzt eine als VWHNF markierte Variable durch einen Term. Der Parameter `force` bestimmt, ob diese Ersetzung erzwungen wird, oder ob die Variable gleich `v` sein muß. Der String `new` ist der Suffix, der zur Umbenennung der Variablen verwendet wird. Ist er leer (`""`) , dann wird nicht umbenannt. Dieses Verhalten ist bei der (*ucp*)-Reduktion erwünscht.

```
replaceMarkedVar force new v e1 e2
    = applyFunc copy e2
        where
          copy e@(EVar VWHNF v')
            | v == v' || force = copyTerm
            | otherwise        = e
          copy e               = e
          copyTerm = if new == ""
                        then e1
                        else renameBoundVars new e1
```

Um die Konvention 3.1.1 für Variablennamen in eingegebenen und generierten Ausrücken einzuhalten, werden alle Variablen eines Ausdruck sukzessiv umbenannt. Dazu werden in einer assoziativen Liste die „alten“ und „neuen“ Namen gesammelt und der aktuelle Name für neue Bindungen mitgeführt (`n`). Trifft die Umbenennungsfunktion `correct'` auf eine Variable, dann wird der Name in der assoziativen Liste gesucht und durch den neuen Namen ersetzt. Falls der Variablenname nicht in der Liste vorhanden ist, dann handelt es sich um eine freie Variable, die nicht umbenannt werden kann.

```
correctCoreExpr e
  = fst (correct' [] "a" e)
      where
        correct' rs n (EVar m var)
            = (EVar m var', n)
              where
                var' = case [v'' | (v', v'') <- rs, v' == var]
                         of [] -> var
                            vs -> head vs
```

Ein Konstruktor enthält keine Variable und bleibt daher wie er ist.

```
        correct' rs n (ECons m t a)
            = (ECons m t a, n)
```

Durch eine Abstraktion wird eine neue Variable eingeführt, die innerhalb des Körpers des λ-Ausdrucks umbenannt werden muß. Dazu wird das Paar `(v,n)` an den Anfang der Assoziationsliste gestellt und `correct'` für `e` aufgerufen. Das Ergebnis der Funktion ist ein umbenannter Ausdruck und der nächste mögliche bzw. unbenutzte Variablenname, der „nach oben“ weitergereicht wird.

```
        correct' rs n (ELam _ v e)
            = (ELam Unmarked n e', n')
              where
                (e', n') = correct' ((v, n):rs) (nextVar n) e
```

Beim `choice`-Ausdruck und Applikationen werden nacheinander die beiden Teilterme umbenannt und der nächstmögliche Name weitergegeben.

```
        correct' rs n (EChoice m e1 e2)
            = (EChoice m e1' e2', n2')
              where
                (e1', n1') = correct' rs n   e1
                (e2', n2') = correct' rs n1' e2

        correct' rs n (EApp m e1 e2)
            = (EApp m e1' e2', n2')
              where
                (e1', n1') = correct' rs n   e1
                (e2', n2') = correct' rs n1' e2
```

Die Umbenennung von `let(rec)`-Ausdrücken gestaltet sich da schon etwas schwieriger. Zuerst müssen alle Bindungsvariablen gesammelt und mit ihrem neuen Namen versehen in die Assoziationsliste übernommen werden. Dann werden die Ausdrücke der Bindungen nacheinander umbenannt, wobei der nächstmögliche Variablenname immer wieder an den nächsten Umbenennungsschritt weitergegeben werden muß. Dies wird mittels der Standard-Funktion `mapAccumL` realisiert. Zum Schluß muß dann nur noch der Term `e` umbenannt werden.

```
correct' rs n (ELet m rec bs e)
    = (ELet m rec bs'' e', ne')
      where
        rs'         = letrs ++ rs
        (n',letrs) = let f = \x y -> (nextVar x, (y,x))
                         in mapAccumL f n [v | (v,_) <- bs]
        bs'         = zip (snd (unzip letrs)) [e | (_,e) <- bs]
        (n'',bs'') = let f = \x (v,e) -> let (e',x') = correct' rs' x e
                                             in (x',(v,e'))
                         in mapAccumL f n' [b' | b' <- bs']
        (e', ne')  = correct' rs' n'' e
```

Nicht minder kompliziert ist das Umbenennen von `case`-Ausdrücken, das mit der Umbenennung des zu überprüfenden Ausdrucks beginnt. Danach werden alle Alternativen parallel umbenannt, denn nach einer `case`-Reduktion wird nur genau eine Alternative weiterverwendet, so daß sich die Variablennamen überschneiden können. Daher muß auch das Maximum der neuen Namen zurückgegeben.

```
correct' rs n (ECase m t e as)
    = (ECase m t e' as', n')
      where
        (e', n'') = correct' rs n e
        as'        = fst as''
        n'         = maximum (snd as'')
        f x y      = (nextVar x, (y,x))
        as''       = unzip [ ((an, avs', ae'), na)
                           | (an, avs, ae) <- as
                           , let (nv,avs'') = mapAccumL f n'' avs
                           , let rs'         = avs'' ++ rs
                           , let avs'        = snd (unzip avs'')
                           , let (ae', na)  = correct' rs' nv ae]
```

A.2 Modul *Reduction*

Die Funktion `getReductions` liefert alle möglichen Reduktionsfolgen eines Ausdrucks. Es werden schrittweise alle möglichen NO- und I-Redexe reduziert und die resultierenden Redukte rekursiv der Funktion übergeben. Gleichzeitig wird jeder Reduktionspfad an jeden reduzierten Ausdruck gebunden. Über die Parameter der Funktion können unerwünschte Pfade ausgeschlossen werden, so z. B. gemischte Folgen von NO- und I-Reduktionen.

```
getReductions rec ireds i_no no_i exprs
    = []:[ path | (path,_) <- sim 1 [([],exprs)]]
        where
          sim n xs = case concat [ step n rs e
                                 | (rs, es) <- xs
                                 , e        <- es
                                 ]
                       of []    -> []
                          paths -> paths ++ (sim (n + 1) paths)
```

Alle Reduktionsschritte für einen einzelnen Ausdruck.

```
          step n rs e
            = [(rs ++ [r], es) | (r,es) <- reds]
              where
                reds    = case markNORedex rec e
                            of HasRedex r e' -> (redsNO r e') ++ (redsI (/= e'))
                               otherwise     -> redsI (@x -> True)
```

Eine (*nd*)-Reduktion muß in die zwei Reduktionen (*ndl*) und (*ndr*) aufgeteilt werden.

```
                redsNO ND e' = [ ((NO r'), reduceCoreExpr rec n e'')
                               | i_no || all isNOreduction rs
                               , r' <- [NDL,NDR]
                               , let e'' = replaceND r' e'
                               ]
                redsNO r  e' = [ ((NO r), reduceCoreExpr rec n e')
                               | i_no || all isNOreduction rs
                               ]
```

Die internen Reduktionen können sowohl mehrfach vorkommen als auch mehrere Redukte liefern.

```
                redsI f = [ (I t, eIs)
                          | no_i || all isIreduction rs
                          , t <- ireds
                          , let eIs = concat [ reduceCoreExpr rec n eI
                                             | eI <- markRedex rec t e
                                             , f eI
                                             ]
                          , eIs /= []
                          ]
```

A.3 Verschiedene Funktionen

Num-Instanz

Zur Charakterisierung der untersuchten Termklassen verwenden wir ein 5-Tupel, das in Haskell zwar benutzbar ist, beim HBC-Compiler aber nicht automatisch als Instanz der `Num`-Klasse erzeugt wird. Wir definieren die Funktionen `+`, `-`, `*`, `negate`, `signum`, `abs` und `fromInteger` für Paare und 5-Tupel:

```
instance (Num a, Num b) => Num (a, b)
   where
    (a1,b1) + (a2,b2) = (a1 + a2, b1 + b2)
    (a1,b1) - (a2,b2) = (a1 - a2, b1 - b2)
    (a1,b1) * (a2,b2) = (a1 * a2, b1 * b2)
    negate (a,b)      = (negate a,negate b)
    signum (a,b)      = (signum a,signum b)
    abs    (a,b)      = (abs    a,abs    b)
    fromInteger n     = (fromInteger n,fromInteger n)

instance (Num a, Num b, Num c, Num d, Num e) => Num (a, b, c, d, e)
   where
    (a1,b1,c1,d1,e1) + (a2,b2,c2,d2,e2) = (a1+a2, b1+b2, c1+c2, d1+d2, e1+e2)
    (a1,b1,c1,d1,e1) - (a2,b2,c2,d2,e2) = (a1-a2, b1-b2, c1-c2, d1-d2, e1-e2)
    (a1,b1,c1,d1,e1) * (a2,b2,c2,d2,e2) = (a1*a2, b1*b2, c1*c2, d1*d2, e1*e2)
    negate (a,b,c,d,e) = (negate a,negate b,negate c,negate d,negate e)
    signum (a,b,c,d,e) = (signum a,signum b,signum c,signum d,signum e)
    abs    (a,b,c,d,e) = (abs    a,abs    b,abs    c,abs    d,abs    e)
    fromInteger n
      = (fromInteger n,fromInteger n,fromInteger n,fromInteger n,fromInteger n)
```

Kreuzprodukte

```
cross as bs
    = [(a, b)| a <- as, b <- bs]

cross5 as bs cs ds es
    = [(a, b, c, d, e) | a <- as, b <- bs, c <- cs, d <- ds, e <- es]
```

Die oben angegebenen Funktionen setzen allerdings vorraus, daß die Mengen endlich sind, da sonst Endlos-Schleifen in den übergeordneten Funktionen entstehen können. Aus diesem Grund definieren wir eine Funktion zum Erzeugen des Kreuzprodukts zweier Mengen mittels Diagonalisierung.

```
diag xs ys
    = diagonalise' 1 xs ys
        where
          diagonalise' n xs ys
            = case diagonal n xs ys
                of [] -> []
                   ds -> ds ++ diagonalise' (n + 1) xs ys

          diagonal n xs ys
            = zip (f ys' xs') (reverse (f xs' ys'))
              where
                xs'     = take n xs
                ys'     = take n ys
                f as bs = if n > length as
                             then drop (n - length as) bs
                             else bs
```

Die Bindungen der `letrec`-Ausdrücke können in beliebiger Reihenfolge vorkommen. Zum Vergleich zweier `letrec`-Ausdrücke müssen also alle Permutationen überprüft werden (siehe `Eq`-Klasse).

```
permut []     = [[]]
permut (x:xs) = concatMap insert (permut xs)
                 where
                   insert []          = [[x]]
                   insert ys'@(y:ys) = (x:ys') : map (y:) (insert ys)
```

Anhang B

Standard-Prelude

Die beiden Datei *prelude.core* und *rules.rule* enthalten die vordefinierten Datentypen, Funktionen und Reduktionsdiagramme für Jonah. Eigene Definitionen können in diese Dateien einfließen oder in separate Dateien ausgelagert werden und Jonah mit den Optionen `--file` und `--rulefile` übergeben werden (siehe Anhang C).

B.1 *prelude.core*

In der Datei *prelude.core* sind beispielhaft einige Funktionen in der von uns verwendeten Kernsprache definiert. Die Funktionen können im interaktiven Teil von Jonah z. B. schrittweise reduziert werden. Jede *prelude*-Funktion kann auch zur Validierung von Reduktionsdiagrammen verwendet werden. Weiterhin müssen in der Datei *prelude.core* die Datentypen angegeben werden, die während einer Validierung betrachtet werden sollen.

Abstrakte Datentypen

```
data List = Nil
          | Cons x xs;

data Bool = True
          | False;

data Nat  = Zero
          | Succ x;
```

Funktionen

Bottom ($\perp$)

```
bot = bot;
```

Identität

```
id x = x;
```

K, *S* und *twice*

```
k x y  = x;
k1 x y = y;

s f g x = f x (g x);

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

Y-Kombinator für Rekursion

```
recursion = \f.(\x.(f (x x))) (\x.(f (x x)));
```

Boole'sche Operatoren

```
not x   = case x
            of True  -> False,
               False -> True;

and x y = case x
            of True  -> (case y
                           of True  -> True,
                              False -> False),
               False -> False;

or  x y = case x
            of True  -> True,
               False -> (case y
                           of True  -> True,
                              False -> False);
```

Listen

```
empty l = case l
            of Nil       -> True,
               Cons x xs -> False;

repeat xx = Cons xx (repeat xx);

reverse a = case a
              of Nil       -> Nil,
                 Cons x xs -> append (reverse xs) (Cons x Nil);
```

```
length l = case l
             of Nil       -> Zero,
                Cons x xs -> Succ (length xs);

length'       = recursion length_;
length_ len l = case l
                  of Nil       -> Zero,
                     Cons x xs -> Succ (len xs);


head x = case x
           of Nil       -> False,
              Cons y ys -> y;

last a = case a
           of Nil       -> False,
              Cons x xs -> (case xs
                              of Nil       -> x,
                                 Cons y ys -> last ys);


tail x = case x
           of Nil       -> False,
              Cons y ys -> ys;

take n l = case n
             of Zero   -> Nil,
                Succ x -> (case l
                             of Nil      -> Nil,
                                Cons y z -> Cons y (take x z));


enum a b = case (nGt a b)
             of True  -> Nil,
                False -> (case (nEq a b)
                            of True  -> Cons a Nil,
                               False -> Cons a (enum (add n1 a) b));

enumFrom a = Cons a (enumFrom (add n1 a));

append l1 l2 = case l1
                 of Nil      -> l2,
                    Cons x y -> Cons x (append y l2);
```

Gleichheit von Zahlen-Listen

```
nListEq l1 l2 = case l1
                  of Nil       -> (case l2
                                     of Nil       -> True,
                                        Cons y ys -> False),
                     Cons x xs -> (case l2
                                      of Nil       -> False,
                                         Cons y ys -> and (nEq x y)
                                                          (nListEq xs ys));
```

map

```
map f l = case l
            of Nil       -> Nil,
               Cons x xs -> Cons (f x) (map f xs);
```

Peano-Zahlen

```
n0  = Zero;
n1  = Succ Zero;
n2  = Succ (Succ Zero);
n3  = Succ (Succ (Succ Zero));
n4  = Succ (Succ (Succ (Succ Zero)));
n5  = Succ (Succ (Succ (Succ (Succ Zero))));
n6  = Succ (Succ (Succ (Succ (Succ (Succ Zero)))));
n7  = Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero))))));
n8  = Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))));
n9  = Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero))))))));
n10 = Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))));
```

Listen von Peano-Zahlen

```
nl0 = Nil;
nl1 = Cons Zero Nil;
nl2 = Cons Zero (Cons Zero Nil);
nl3 = Cons Zero (Cons Zero (Cons Zero Nil));
nl4 = Cons Zero (Cons Zero (Cons Zero (Cons Zero Nil)));
nl5 = Cons Zero (Cons Zero (Cons Zero (Cons Zero (Cons Zero Nil))));

isZero x = case x
             of Zero   -> True,
                Succ y -> False;
```

Vergleiche von Peano-Zahlen

```
nEq a b = case a
            of Zero   -> (case b
                            of Zero   -> True,
                               Succ y -> False),
               Succ x -> (case b
                            of Zero   -> False,
                               Succ y -> nEq x y);


-- ‘<‘
nLt a b = case a
            of Zero   -> (case b
                            of Zero   -> False,
                               Succ y -> True),
               Succ x -> (case b
                            of Zero   -> False,
                               Succ y -> nLt x y);
```

```
-- `>`
nGt a b = case a
            of Zero   -> (case b
                           of Zero   -> False,
                              Succ y -> False),
               Succ x -> (case b
                           of Zero   -> True,
                              Succ y -> nGt x y);


-- `<=`
nLe a b = or (nLt a b) (nEq a b);

-- `>=`
nGe a b = or (nGt a b) (nEq a b);
```

Addition, Subtraktion und Multiplikation von Peano-Zahlen

```
add a b = case a
            of Zero   -> b,
               Succ x -> add x (Succ b);

double x = add x x;

mult a b = case a
             of Zero   -> Zero,
                Succ x -> add (mult x b) b;

sub a b = case b
            of Zero   -> a,
               Succ x -> (case a
                            of Zero   -> False,
                               Succ y -> sub y x);
```

Fakultät

```
fac n = case n
          of Zero   -> Succ Zero,
             Succ m -> mult n (fac m);

recfac n = letrec fac' = (\x -> case x
                                  of Zero   -> Succ Zero,
                                     Succ y -> mult x (fac' y))
               in fac' n;
```

B.2 *rules.rule*

Die in dieser Arbeit angegebenen Reduktionsdiagramme bzw. Diagrammsätze sind in der Datei *rules.rule* in der von uns gewählten Notation definiert. Aus Platzgründen sollen exemplarisch nur die Diagrammsätze der (*llet*)-, (*ldel*)- und (*lcom*)-Reduktion angegeben werden.

Reduktionsdiagramme für (*llet*) von Seite 96

```
> llet:

> i.llet n.a ==> n.a i.llet
  where
    a <- {lbeta, abs, lapp, llet, cp, lcase, case, ndl, ndr}

> i.llet n.a ==> n.a n.llet
  where
    a <- {lbeta, abs, lapp, llet, cp, lcase, case, ndl, ndr}

> i.llet {n.llet, n.lapp, n.lcase}+ ==> {n.llet, n.lapp, n.lcase}+ (i.llet)?
```

Reduktionsdiagramme für (*ldel*) von Seite 100

```
> ldel:

> i.ldel n.a ==> n.a i.ldel
  where
    a <- {lbeta, abs, lapp, llet, cp, lcase, case, ndl, ndr}

> i.ldel n.a ==> {n.lapp, n.llet, n.lcase}* n.a i.ldel
  where
    a <- {lbeta, case, cp, ndl, ndr, abs}

> i.ldel n.a ==> {n.lapp, n.llet, n.lcase}* i.ldel
  where
    a <- {lapp, llet, lcase}
```

Reduktionsdiagramme für (*lcom*) von Seite 104

```
> lcom:

> i.lcom n.a ==> n.a i.lcom
  where
    a <- {lbeta, abs, lapp, llet, cp, lcase, case, ndl, ndr}

> i.lcom {n.llet, n.lapp, n.lcase}+ ==> {n.llet, n.lapp, n.lcase}* i.lcom

> i.lcom {n.llet, n.lapp, n.lcase}+ ==> {n.llet, n.lapp, n.lcase}* i.lcom i.lcom
```

Anhang C

Jonah-Manual

Der Aufruf von JONAH ist allgemein[1]:

```
$ jonah +RTS run time flags -RTS command [options]
```

Jedem Befehl (*command*) können (bzw. müssen) weitere Parameter übergeben werden.

Der `help`-Befehl

Dieser Befehl zeigt alle möglichen Befehle an. Wenn zusätzlich noch einer der Befehle angegeben wird, dann werden alle Optionen und Parameter für diesen speziellen Befehl ausgegeben.

```
$ jonah -RTS help
Usage: jonah [+RTS run time flags] -RTS command [options]
 commands:
   help                                 Help
   cgi                                  Work as CGI script
   applyrule                            Apply rule to term
   count                                Count expressions of class
   deduce                               Deduce rules for term
   evaluate                             Evaluate term
   generate                             Generate terms
   validate                             Validate rules for terms
 options for ‘help‘:
   [One of the commands from above]
```

Der `version`-Befehl

Mit diesem Befehl werden die Version und das Kompilierdatum von JONAH sowie die benutzte Compiler-Version ausgegeben. Zusätzlich kann in der Datei `MainAll.hs` ein Zusatz definiert werden, der die Programmversion genauer bezeichnen kann (hier `(letrec-case)`).

[1] Die Optionen `+RTS` und `-RTS` umschließen in den benutzten Compiler-Versionen (GHC 4.06 und HBC 0.9999.5b) die Laufzeit-Einstellungen.

```
$ jonah -RTS version
Jonah 0.8.2 (letrec-case) [2000-07-16 19:37:07]
hbc Haskell98 version 0.9999.5b, 1999 Apr 02
```

Der cgi-Befehl

Dieser Befehl benötigt weder Optionen noch Parameter, da diese auf der generierten HTML-Seite einstellbar sind. Der CGI-Modus von JONAH stellt allerdings nicht den gesamten Funktionsumfang bereit. Er sind nur die Funktionen zur Evaluation von Termen, dem Finden und der Validierung von Reduktionsdiagrammen implementiert. Von der Validierung sollte jedoch abgesehen werden, da die Analyse sehr viel Rechenkapazität in Form von Zeit und Speicher benötigt, und dadurch der Web-Server zum Stillstand gebracht werden kann. Daher ist eine Beschränkung der Termklasse auf maximal 3 Sprachkonstrukte festgelegt.

Der Aufruf von JONAH als CGI-Programm kann auch dadurch erreicht werden, indem der Programmname mit der Erweiterung *.cgi* endet. Zu beachten ist, daß JONAH dann aber nur noch als CGI-Programm startet. Das CGI-Skript kann unter

`http://www.informatik.uni-frankfurt.de/~mhuber/cgi/jonah.cgi`

ausprobiert werden.

Allgemeine Optionen und Parameter

Die Optionen werden durch zwei Minuszeichen eingeleitet. Durch ein Gleichheitszeichen können der Option Parameter zugewiesen werden. Die Parameter können Listen von z. B. Reduktionstypen sein, die entweder durch Kommata oder Leerzeichen getrennt sein müssen, wobei im letzteren Fall der gesamte Parameter durch Anführungszeichen eingeschlossen werden muß. Wenn Zahlenwerte als Parameter übergeben werden, dann wird der Standardwert bei Aufruf der Hilfe-Seite[2] angezeigt.

`--console`	Die Ausgabe enthält Kontroll-Sequenzen für z. B. Farbe. Dadurch kann bei langen Analysen der Fortschritt besser überwacht werden, oder eine übersichtliche Darstellung der Ergebnisse erreicht werden.
`--file="filename"`	Standardmäßig wird die Datei *prelude.core* mit einigen vordefinierten Funktionen und Datentypen geladen. Zusätzlich kann eine angegebene Datei (`filename`) geladen werden.
`--letrec[=n]`	Umschalten auf den rekursiven Kalkül. Die optionale Angabe der maximal verwendbaren `letrec`-Bindungen beeinflußt die Termgenerierung.
`--maxdepth=n`	Bei der Suche nach neuen Diagrammen wird nach einer definierbaren Suchtiefe abgebrochen.

[2] `jonah -RTS help` bzw. `:Set` im interaktiven Modus.

`--maxgen=n`	Beim Generieren von allgemeinen Diagrammen mit Wiederholungen wird bis zu einer definierbaren Anzahl von Wiederholungen abgebrochen.
`--maxred=n`	Um einen Term auf Terminierung zu überprüfen, wird er bis zu einer einstellbaren Anzahl von Schritten reduziert.
`--noprelude`	Die Datei *prelude.core* wird nicht geladen.
`--norules`	Die Datei *rules.rule* wird nicht geladen.
`--rulefile="filename"`	Statt *rules.rule* wird die Datei `filename` geladen.
`--stats=n`	Während einer Analyse wird alle `n` Schritte ein Zwischenergebnis ausgegeben. Bei gleichzeitiger Option `--console` wird die Statistik immer wieder neu überschrieben.
`--time`	Die Dauer einer Analyse wird angegeben.
`--verbose[=1..3]`	Die Ergebnisse von JONAH können in drei verschiedenen Detail-Stufen ausgegeben werden.

Parameter für den `applyrule`-Befehl

Mit dem `applyrule`-Befehl kann auf einen einzelnen Term ein Reduktionsdiagramm (bzw. -Regel) angewendet werden. Daher sind auch nur zwei Parameter obligatorisch:

`--term="term"`	Der übergebene Term darf kein Loch enthalten, jedoch alle Funktionen und Datentypen, die in der Datei *prelude.core* definiert sind.
`--rule="rule"`	Die Regel (bzw. das Diagramm) wird wie in Kapitel 6.1.1 definiert (siehe Abbildung 6.2, Seite 79), jedoch ohne das einleitende `>`-Zeichen. Es können sowohl Wiederholungs- als auch Mengenkonstrukte verwendet werden.

```
$ jonah -RTS applyrule --term="(let x=let y=Succ in y in x) Zero"
--rule="i.llet n.lapp n.lapp ==> n.lapp n.llet"

Rule: i.llet n.lapp n.lapp ==> n.lapp n.llet
Red.: («llet:(let x=(let y=Succ in y) in x)» Zero)
   L:
     *(let y=Succ in (let x=y in (x Zero)))
   R:
     *(let y=Succ in (let x=y in (x Zero)))
  ==> MATCH
```

Die Terme der Schnittmenge beider Ergebnismengen werden mit einem Stern (`*`) markiert.

Optionen und Parameter für den `count`-Befehl

Der `count`-Befehl zählt alle Terme einer Termklasse inkl. ihrer Term-Subklassen. Dazu werden die Klasse und die Datentypen benötigt:

`--class=c1,c2,...`	Es müssen bis zu fünf positive Zahlen, getrennt durch Kommata, übergeben werden, die die Anzahl der zu verwendenden Sprachkonstrukte repräsentieren (siehe Definition 5.1.2, Seite 65).
`--types=t1,t2,...`	Eine durch Kommata getrennte Liste von Datentypen, die in *prelude.core* definiert sind.
`--classonly`	Nur die Anzahl der Terme der gegebenen Termklasse wird errechnet.

```
$ jonah -RTS count --class=1,1,2,1,1 --types=List --verbose
```

```
20,007,233
```

bzw.

```
$ jonah -RTS count --class=1,1,2,1,1 --types=List --verbose --classonly --time
```

```
17,591,610
Execution time: 2 s
```

Optionen und Parameter für den `deduce`-Befehl

Der `deduce`-Befehl ermittelt für einen Term alle möglichen Reduktionsdiagramme.

`--term="term"`	Der übergebene Term darf kein Loch enthalten, jedoch alle Funktionen und Datentypen, die in der Datei *prelude.core* definiert sind.
`--redexes=r1,r2,...`	Eine durch Kommata getrennte Liste von Reduktionstypen, die in der rechten Folge des Diagramms auftreten dürfen.
`--allrules`	Schaltet die Optimierung aus, so daß auch (vielleicht) irrelevante Diagramme angezeigt werden.

```
$ jonah -RTS deduce --term="(let x=let y=Succ in y in x) Zero"
```

```
llet:
i.llet n.lapp n.lapp ==> n.lapp n.llet
ucp:
i.ucp ==> n.lapp i.ucp
```

Parameter für den `evaluate`-Befehl

Ein Term wird mittels des `evaluate`-Befehls in Normalordnung ausgewertet.

`--term="term"`	Der übergebene Term darf kein Loch enthalten, jedoch alle Funktionen und Datentypen, die in der Datei *prelude.core* definiert sind.

```
$ jonah -RTS evaluate --term="(let x=let y=Succ in y in x) Zero" --verbose=2
```

```
    lapp: ((let x=(let y=Succ in y) in x) Zero)
     1\/  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    llet: (let x=(let y=Succ in y) in (x Zero))
     2\/  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    cp y: (let y=Succ in (let x=[y] in (x Zero)))
     3\/  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    cp x: (let y=Succ in (let x=Succ in ([x] Zero)))
     4\/                 ^^^^^^^^^^^^^^^^^^^^^^^^^^
     abs: (let y=Succ in (let x=Succ in (Succ Zero)))
     5\/                                ^^^^^^^^^^^
(let y=Succ in (let x=Succ in (let new_5=Zero in (Succ new_5))))
(5 reductions)
```

Optionen und Parameter für den `generate`-Befehl

Der `generate`-Befehl listet alle Terme einer Termklasse auf. Wenn ein Term keine terminierende Reduktion besitzt (jedenfalls bis zur einstellbaren Reduktionstiefe) oder schon eine WHNF ist, dann wird er entsprechend markiert.

`--class=c1,c2,...`	Es müssen bis zu fünf positive Zahlen, getrennt durch Kommata, übergeben werden, die die Anzahl der zu verwendenden Sprachkonstrukte repräsentieren (siehe Definition 5.1.2, Seite 65).
`--types=t1,t2,...`	Eine durch Kommata getrennte Liste von Datentypen, die in *prelude.core* definiert sind.
`--classonly`	Nur die Terme der gegebenen Termklasse werden generiert und auf Terminierung bzw. WHNF überprüft.

```
$ jonah -RTS generate --class=0,1,1 --types=List

17 expressions in class (0, 1, 1, 0, 0)
Nil -- no reduction
(let a=Nil in a)
(let a=Nil in Nil) -- no reduction
(let a=Cons in a)
(let a=Cons in Nil) -- no reduction
((let a=Nil in a) Nil) -- no reduction
((let a=Nil in Nil) Nil) -- no reduction
((let a=Cons in a) Nil)
((let a=Cons in Nil) Nil) -- no reduction
(let a=Nil in (a a)) -- no reduction
(let a=Nil in (a Nil)) -- no reduction
(let a=Cons in (a a))
(let a=Cons in (a Nil))
(let a=(Cons Nil) in a)
(let a=(Cons Nil) in Nil) -- no reduction
(let a=(Cons Cons) in a)
(let a=(Cons Cons) in Nil) -- no reduction
```

Optionen und Parameter für den `validate`-Befehl

Der `validate`-Befehl validiert alle angegebenen Diagrammsätze (Regelsätze) für alle Terme der gewählten Termklasse.

`--rulesets=rs1,rs2,...`	Eine durch Kommata getrennte Liste von Diagrammsätzen, die in *rules.rule* definiert sind.
`--redexes=r1,r2,...`	Eine Liste von Reduktionstypen, die in der rechten Folge des Diagramms auftreten dürfen (siehe `deduce`-Befehl).
`--types=t1,t2,...`	Eine Liste von Datentypen, die in *prelude.core* definiert sind.
`--addrules`	Wenn neue Diagramme ermittelt werden konnten, dann bestimmt diese Option, daß die neuen Diagramme dem bisherigen Diagrammsatz hinzugefügt werden.
`--class=c1,c2,...`	Es müssen bis zu fünf positive Zahlen übergeben werden, die die Anzahl der zu verwendenden Sprachkonstrukte repräsentieren (siehe Definition 5.1.2, Seite 65).
`--classonly`	Nur die Terme der gegebenen Termklasse werden generiert und auf Terminierung bzw. WHNF überprüft.
`--ignore=n,rs:t:n,...`	Es werden die ersten `n` Terme übersprungen. Falls die Statistik-Option `--stats` gewählt ist, wird dieser Parameter im ensprechenden Zyklus ausgegeben, so daß nach einem Abbruch mit der Validierung an dieser Stelle fortgefahren werden kann. (Die Werte `t` und `n` sind die für den Diagrammsatz `rs` bisher getesteten bzw. negativ validierten Terme)

Der erste Parameter zeigt, daß JONAH mehrere Diagrammsätze gleichzeitig validieren kann. Das hat den einfachen Grund, daß die eigentliche Validierung der Diagramme für einen einzelnen Term nicht sehr zeitintensiv ist, die Erzeugung der Terme dagegen schon. So können in einem Durchlauf alle Terme der Termklasse für alle Diagrammsätze geprüft werden. Die in Kapitel 7 gezeigten Anwendungsbeispiele geben einen Eindruck, welche Parameter in welchen Fällen benutzt werden können.

Bemerkungen

- Bei umfangreichen Testläufen ist es ratsam die Statistik-Option `--stats=n` zu verwenden, da es zu Speichermangel kommen kann. Die Ausgabe von Zwischenergebnissen kann — wegen der folgenden *garbage collection* — wieder Speicherplatz freigeben.

- Es sollte von der Möglichkeit Gebrauch gemacht werden, mehrere Diagrammsätze gleichzeitig zu validieren, um Rechenzeit zu sparen.

- Da die Anzahl der generierten Terme im rekursiven Kalkül sehr groß werden kann, sollte überlegt werden, gezielt schwierige Kontexte zu überprüfen. D. h. die Terme in einen konstruierten Kontext zu setzen, wie etwa `(letrec x=(letrec y=[] in Cons y),... in case x of ...)`.

Jonah interaktiv

Wenn JONAH ohne Befehl aufgerufen wird, dann kann das System interaktiv gesteuert werden. Dieser Modus enthält alle Befehle des nicht-interaktiven Modus, jedoch können die verschiedenen Parameter nicht so fein abgestimmt werden.

```
$ jonah
  _/_/_/_/  _/_/_/  _/    _/     _/     _/    _/            Jonah 0.8.2
       _/ _/    _/ _/_/  _/   _/  _/   _/    _/  validates commutation diagrams
      _/ _/    _/ _/ _/ _/ _/_/_/_/   _/_/_/_/      for a non-deterministic
_/   _/ _/    _/ _/  _/_/ _/      _/ _/    _/          lambda  calculus
 _/_/   _/_/_/  _/    _/ _/      _/ _/    _/             in Haskell 98
(c) 2000 Michael Huber
Type :? for help
Jonah [rec]>
```

Nach dem Aufruf von JONAH können an der Eingabemarke (`Jonah [rec]>`[3]) verschiedene Befehle eingegeben werden, die weitestgehend die Funktionalität der oben beschriebenen besitzen.

Jedoch ist es nur im interaktiven Modus möglich, einen Ausdruck Schritt für Schritt zu reduzieren. Dabei kann man wählen, ob der aktuelle NO-Redex oder ein interner Redex reduziert werden soll.

Ein Beispiel soll den Ablauf etwas verdeutlichen. Wir geben an der Eingabemarke den Befehl `:step` mit einem Ausdruck ein. Der Ausdruck und eine Liste von verschieden markierten Redexen werden ausgegeben.

```
Jonah [rec]> :st (letrec x=(\a.a) in x) Nil
[Entering interactive reduction mode]
Expression: ((letrec x=(\a.a) in x) Nil)
+:NO      lapp: ((letrec x=(\a.a) in x) Nil)
a:I       cp/x: ((letrec x=(\a.a) in [x]) Nil)
b:I      cpt/x: ((letrec x=(\a.a) in [x]) Nil)
c:I     ldup/x: ((letrec x=(\a.a) in x) Nil)
d:I      ucp/x: ((letrec x=(\a.a) in [x]) Nil)
1. Choices: [-] stop, [+] normal order, [a,b,c,d] internal ? +
```

JONAH unterlegt jeden Redex grau. Hier soll dies durch die Unterstreichung angedeutet werden. Alle Reduktionsschritte werden numeriert und man hat die Auswahl, das Reduzieren zu verlassen, einen No-Redex oder internen Redex zu reduzieren. Wir wählen den NO-Redex, der das `letrec` „nach außen“ schiebt.

```
+:NO       cp/x: (letrec x=(\a.a) in ([x] Nil))
a:I      ldup/x: (letrec x=(\a.a) in (x Nil))
b:I       ucp/x: (letrec x=(\a.a) in ([x] Nil))
2. Choices: [-] stop, [+] normal order, [a,b] internal ? +
```

[3] Anhand der Eingabemarke wird dem Benutzer der aktuelle Kalkül-Modus angezeigt, hier ist der rekursive Kalkül eingestellt. Mit dem Befehl `:set -r` wird auf den nicht-rekursiven Kalkül geschaltet.

Die (*cp*)-Reduktion kopiert die Abstraktion in die Anwendung `(x Nil)`. Dabei werden die gebundenen Variablen (in diesem Fall `a`) derart umbenannt, daß keine doppelten Namen entstehen, da sonst unsere Variablennamenkonvention verletzt würde. An jede gebundene Variable wird einfach, durch einen Unterstrich getrennt, der aktuelle Reduktionszähler angehängt. Dies ist ohne weiteres möglich, da wir nur intern, also während der Reduktion, Variablen dieses Formats erlauben. D. h. der Parser würde den Ausdruck `letrec x_4 = Nil in x_4` abweisen (siehe Kapitel 3.1.1).

```
+:NO     lbeta: (letrec x=(\a.a) in ((\a_2.a_2) Nil))
a:I     ldel/x: (letrec x=(\a.a) in ((\a_2.a_2) Nil))
b:I    ldels/x: (letrec x=(\a.a) in ((\a_2.a_2) Nil))
c:I ldelcyc1/x: (letrec x=(\a.a) in ((\a_2.a_2) Nil))
d:I   ldelcyc2: (letrec x=(\a.a) in ((\a_2.a_2) Nil))
e:I     ldup/x: (letrec x=(\a.a) in ((\a_2.a_2) Nil))
3. Choices: [-] stop, [+] normal order, [a,b,c,d,e] internal ? +
```

Die (*lbeta*)-Reduktion bindet das Argument `Nil` an die Variable `a_2` zu einem neuen `letrec`-Ausdruck.

```
+:NO    lletr1: (letrec x=(\a.a) in (letrec a_2=Nil in a_2))
a:I     cp/a_2: (letrec x=(\a.a) in (letrec a_2=Nil in [a_2]))
b:I    cpt/a_2: (letrec x=(\a.a) in (letrec a_2=Nil in [a_2]))
c:I     ldel/x: (letrec x=(\a.a) in (letrec a_2=Nil in a_2))
d:I    ldels/x: (letrec x=(\a.a) in (letrec a_2=Nil in a_2))
e:I ldelcyc1/x: (letrec x=(\a.a) in (letrec a_2=Nil in a_2))
f:I   ldelcyc2: (letrec x=(\a.a) in (letrec a_2=Nil in a_2))
g:I     ldup/x: (letrec x=(\a.a) in (letrec a_2=Nil in a_2))
h:I   ldup/a_2: (letrec x=(\a.a) in (letrec a_2=Nil in a_2))
i:I    ucp/a_2: (letrec x=(\a.a) in (letrec a_2=Nil in [a_2]))
4. Choices: [-] stop, [+] normal order, [a,b,c,d,e,f,g,h,i] internal ? +
```

Die beiden gestaffelten `letrec` werden zusammengezogen.

```
+:NO    cp/a_2: (letrec x=(\a.a), a_2=Nil in [a_2])
a:I     ldel/x: (letrec x=(\a.a), a_2=Nil in a_2)
b:I    ldels/x: (letrec x=(\a.a), a_2=Nil in a_2)
c:I ldelcyc1/x: (letrec x=(\a.a), a_2=Nil in a_2)
d:I     ldup/x: (letrec x=(\a.a), a_2=Nil in a_2)
e:I   ldup/a_2: (letrec x=(\a.a), a_2=Nil in a_2)
f:I    ucp/a_2: (letrec x=(\a.a), a_2=Nil in [a_2])
5. Choices: [-] stop, [+] normal order, [a,b,c,d,e,f] internal ? +
```

Nun kann die Variable `a_2` kopiert werden.

```
a:I      ldel/x: (letrec x=(\a.a), a_2=Nil in Nil)
b:I    ldel/a_2: (letrec x=(\a.a), a_2=Nil in Nil)
c:I     ldels/x: (letrec x=(\a.a), a_2=Nil in Nil)
d:I   ldels/a_2: (letrec x=(\a.a), a_2=Nil in Nil)
e:I  ldelcyc1/x: (letrec x=(\a.a), a_2=Nil in Nil)
f:Ildelcyc1/a_2: (letrec x=(\a.a), a_2=Nil in Nil)
g:I    ldelcyc2: (letrec x=(\a.a), a_2=Nil in Nil)
h:I      ldup/x: (letrec x=(\a.a), a_2=Nil in Nil)
i:I    ldup/a_2: (letrec x=(\a.a), a_2=Nil in Nil)
6. Choices: [-] stop, [a,b,c,d,e,f,g,h,i] internal ? a
```

Der Ausdruck enthält keinen NO-Redex mehr, so daß wir den ersten internen Redex, einen (*ldel*)-Redex wählen.

```
a:I    ldel/a_2: (letrec a_2=Nil in Nil)
b:I   ldels/a_2: (letrec a_2=Nil in Nil)
c:Ildelcyc1/a_2: (letrec a_2=Nil in Nil)
d:I    ldelcyc2: (letrec a_2=Nil in Nil)
e:I    ldup/a_2: (letrec a_2=Nil in Nil)
7. Choices: [-] stop, [a,b,c,d,e] internal ? a
```

Auch die Variable `a_2` kann gelöscht werden.

```
Expression: ((letrec x=(\a.a) in x) Nil)
Reduction sequence:
1) n.lapp
2) n.cp
3) n.lbeta
4) n.lletr1
5) n.cp
6) i.ldel
7) i.ldel
Result: Nil
[Leaving interactive reduction mode]
Jonah [rec]>
```

Da der Ausdruck nun nicht weiter reduziert werden kann, wird der Einzelschrittmodus verlassen. Dabei wird die geasmte Reduktionsfolge und das Result ausgegeben.

Die Reduktionen (*nd*) und (*ldup*) liefern mehr als ein mögliches Ergebnis, weshalb wir eines auswählen können. Bei (*nd*) bedeutet dies eine Entscheidung zwischen (*ndl*) und (*ndr*).

```
Jonah [rec]> :step choice True False
[Entering interactive reduction mode]
Expression: (choice True False)
+:NO        nd: (choice True False)
1. Choices: [-] stop, [+] normal order ? +

a: True
b: False
Choose expression: a

Expression: (choice True False)
Reduction sequence:
1) n.nd
Result: True
[Leaving interactive reduction mode]
Jonah [rec]>
```

Anhang D

Erweiterungen für Jonah

Für eigene Erweiterungen soll eine kurze Check-Liste dienen, die die einzelnen Schritte an einem Beispiel aufzeigt. Wir erfinden eine Reduktionsregel (new)[1], die folgendermaßen definiert ist:

Definition D.0.1 Seien C,D beliebige Kontexte. Die (new)-Reduktion identifiziert mindestens zwei syntaktisch gleiche Bindungen x_1 = s_1 bis x_n = s_n, ersetzt alle Vorkommen der x_i durch x und löscht dann für alle $1 \leq i \leq n$ die Bindungen x_i = s_i:

$$C[\,\texttt{letrec}\ x = s, x_1 = s_1, \ldots, x_n = s_n, \ldots\ \texttt{in}\ t\,]$$
$$\xrightarrow{new}\quad C[\,\texttt{letrec}\ x = s, \ldots\ \texttt{in}\ t^{[x\,/\,x_i]}\,]$$

falls für alle $1 \leq i \leq n$ gilt: $s \equiv s_i$ sowie $x \notin \mathcal{FV}(s_i)$ und $x_i \notin \mathcal{FV}(s)$.

Die zwei Dateien *Expression.hs* und *Reduction.hs* müssen für die Reduktionsregel (new) folgendermaßen geändert werden:

Expression.hs

1. Erweiterung des Datentyps *ReductionType*:

```
data ReductionType
     = VAR String
     ...
     | EQL String
     | NEW
     deriving (Ord)
```

   Der interne Name der Reduktionsregel kann beliebig sein, muß allerdings der Namenskonvention eines Haskell-Konstruktors genügen.

[1] Die Reduktion soll hier nur als Beispiel dienen. Ob sie einen Sinn hat sei dahingestellt.

2. Erweiterung der `Eq`-Instanz von `ReductionType` zum Vergleichen des neuen Reduktionstyps:

```
instance Eq ReductionType
    where
      (==) (VAR      a)  (VAR      b) = a == b
      ...
      (==) (EQL      a)  (EQL      b) = a == b
      (==) (NEW       )  (NEW       ) = True
      (==) x             y            = False
```

   Enthält der neue Konstruktor Parameter, dann muß natürlich entschieden werden, ob diese zum Vergleich herangezogen werden müssen (siehe `VAR` bzw. `EQL`).

3. Erweiterung des Parameter-unabhängigen Vergleichs in der Funktion `isSameReductionType`:

```
isSameReductionType (VAR      a)  (VAR      b) = True
...
isSameReductionType (EQL      a)  (EQL      b) = True
isSameReductionType (NEW       )  (NEW       ) = True
isSameReductionType x             y            = False
```

4. Erweiterung der `Show`-Instanz und der Parameter-abhängigen Ausgabe:

```
instance Show ReductionType
    where
      showsPrec _ = showString . toString
        where
          toString a
            = case a
                of VAR v      -> v
                   ...
                   EQL      v -> "eql"
                   NEW      v -> "new"
```

   Hier wird definiert, wie der interne Name in Jonah ausgegeben wird und in der Regeldatei zu verwenden ist. Die Funktion `showReductionType` wird verwendet, um die Parameter-abhängigen Reduktionen genauer zu beschreiben. Wir unterscheiden z. B. das allgemeine (*eql*) vom speziellen, indem der erste Parameter leer ist (`""`). Im Beispiel (*new*) ist das nicht nötig, die Programmzeile kann also auch entfallen:

```
showReductionType
    = showreductiontype
        where
          s = "/"
          showreductiontype r@(VAR v)       = v
          ...
          showreductiontype r@(EQL      "") = show r
          showreductiontype r@(EQL       v) = show r ++ s ++ v
          showreductiontype r@(NEW        ) = show r
          showreductiontype r               = show r
```

5. Die neue Reduktionsregel wird Jonah als Erweiterung bekanntgegeben:

```
extReductionTypes  = [ CopyD    ""
                     ...
                     , EQL      ""
                     , NEW
                     ]
```

Reduction.hs

1. Erweiterung der allgemeine Markierungsfunktion `markRedex'`:

   ```
   markRedex' s@(lam,cons) rec redexType
       = case redexType
           of ----------- Basisregeln ------------
              LBeta      -> markContextR markLBeta
              ...
              -------- Erweiterungsregeln --------
              LDel     v -> markContextR markLDel
              ...
              EQL      p -> markContextC markEQL (True,True)
              NEW        -> markContextS markNEW
   ```

   Hier wird definiert, in welchem Kontext ein (*new*)-Redex gesucht werden soll, z. B. in einem Oberflächenkontext.

2. Die spezielle Markierungsfunktion `markNEW` wird dann entsprechend definiert — auf Effizienz soll hier nicht geachtet werden:

   ```
   markNEW (ELet _ False bs expr) -- (1)
       = []
   markNEW (ELet _ rec@True bs expr) -- (2)
       = [ ELet (Redex NEW) rec bs expr
         | 1 < length [ (v1,v2)
                      | (v1,ve1) <- bs
                      , (v2,ve2) <- bs
                      , v1 /= v2 && ve1 == ve2
                      , v1 `notElem` getFreeVars ve2
                      , v2 `notElem` getFreeVars ve1
                      ]
         ]
   markNEW expr = [] -- (3)
   ```

   In diesem Beispiel können nur `letrec`-Ausdrücke einen (*new*)-Redex enthalten (2), das nicht-rekursive `let` wird nicht betrachtet (1). Sind mindestens zwei Bindungen gleich (α-konvertibel), dann wird der `letrec`-Ausdruck als (*new*)-Redex markiert, alle anderen Terme werden nicht markiert (3). Das Ergebnis der Markierungsfunktion ist eine Liste von unterschiedlich markierten Termen. In diesem Fall ist das nur ein Term.

3. Der markierte Term muß natürlich noch reduziert werden können. Dazu wird die Funktion `reduceCoreExpr` entsprechend erweitert:

   ```
   reduceCoreExpr rec level (ELet (Redex NEW) r bs e)
       = [ELet Unmarked rec bs' e']
         where
           bs' = removeTwoEqualTerms
           e'  = renameRemovedVariables
           removeTwoEqualTerms    = ...
           renameRemovedVariables = ...
   ```

   Eine Unterscheidung zwischen rekursivem und nicht-rekursivem `let` ist hier nicht nötig, da die `let`-Terme erst gar nicht markiert werden. Ansonsten muß dies bei der Reduktion beachtet werden.

Bemerkung. Bei Reduktionen, die `let(rec)`-Bindungen bearbeiten — (*lletr1*), (*cp*), (*ucp*), (*ldel*) usw. — ist es ratsam, den entsprechenden Variablennamen in den `ReductionsType` zu integrieren. Dadurch kann gezielt reduziert werden, ohne den gesamten Term nochmals zu durchlaufen, nur um die entsprechende Variable zu identifizieren. In unserem Beispiel ist das nicht erforderlich, da nur ein Redex markiert wird, und beim Reduzieren die doppelten Bindungen entfernt werden.

Literaturverzeichnis

[Abr90] Abramsky, Samson: *The Lazy lambda-calculus*, Seiten 65 – 117. Addison Wesley, 1990.

[AFM+95] Ariola, Zena M., Matthias Felleisen, John Maraist, Martin Odersky und Philip Wadler: *A Call-By-Need Lambda Calculus*. In: *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, Seiten 233 – 246, New York, NY, Januar 1995. ACM.

[AK96] Ariola, Zena M. und Jan Willem Klop: *Lambda calculus with explicit recursion*. In: *532*, Seite 75. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 31. Dezember 1996. AP (Department of Software Technology).

[Bab87] Baber, Robert Laurence: *The Spine of Software*. John Wiley & Sons, Chichester, 1987.

[Bac78] Backus, J.: *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM, 21:613 –, 1978.

[Bar84] Barendregt, Hendrik Pieter: *The Lambda Calculus - Its Syntax and Semantics*, Band 103 der Reihe *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised Auflage, 1984.

[BW88] Bird, Richard und Philip Wadler: *Introduction to Functional Programming*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1988. Japanese translation, 1991. Dutch translation, 1991. German translation, 1992.

[Dav92] Davie, A. J. T.: *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, Cambridge, 1992.

[HAB+99a] Hudak, Paul, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman und Philip Wadler: *Haskell 98: A Non-strict, Purely Functional Language*. `http://www.haskell.org/onlinereport`, Februar 1999.

[HAB+99b] Hudak, Paul, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman und Philip Wadler: *Haskell 98 Libraries: Standard Libraries for Haskell 98.* `http://www.haskell.org/onlinelibrary`, Februar 1999.

[How89] Howe, Douglas J.: *Equality In Lazy Computation Systems.* In: *Proceedings, Fourth Annual Symposium on Logic in Computer Science* [IEE89], Seiten 198 – 203.

[HPF00] Hudak, Paul, John Peterson und Joseph Fasel: *A Gentle Introduction To Haskell Version 98.* `http://www.haskell.org/tutorial`, Juni 2000.

[Hug89] Hughes, John: *Why Functional Programming Matters.* Computer Journal, 32(2):98 – 107, 1989.

[Hug96] Hughes, John: *Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference.* In: Danvy, Olivier, Robert Glück und Peter Thiemann (Herausgeber): *Partial Evaluation*, Band 1110 der Reihe *LNCS*. Springer-Verlag, Februar 1996.

[IEE89] IEEE Computer Society Press: *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, Asilomar Conference Center, Pacific Grove, California, 5. – 8. Juni 1989.

[Jon94] Jones, Mark P.: *The implementation of the Gofer functional programming system.* Technischer BerichtYALEU/DCS/RR-1030, Department of Computer Science, Yale University, New Haven, Connecticut, USA, May 1994, Mai 1994.

[Jon95] Jones, Mark P.: *From Hindley-Milner Types to First-Class Structures.* In: *Proceedings of the Haskell Workshop*, La Jolla, California, Juni 1995. Yale University Research Report YALEU/DCS/RR-1075.

[Jon00] Jones, Mark P.: *Hugs 98*, Februar 2000. `http://haskell.org/hugs`.

[Klo97] Klose, Norbert: *Lucky Manual*, Version 0.3 Auflage, 1997.

[KSS99] Kutzner, Arne und Manfred Schmidt-Schauss: *A Non-Deterministic Call-by-Need Lambda Calculus.* In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, Band 34(1) der Reihe *ACM SIGPLAN Notices*, Seiten 324 – 335. ACM, Juni 1999.

[Kut00] Kutzner, Arne: *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic Choice: Operationale Semantik, Programmtransformationen und Anwendungen.* Doktorarbeit, Johann Wolfgang Goethe-Universität, Frankfurt am Main, April 2000.

[Les88] Lester, David R.: *Combinator Graph Reduction: A Congruence and its Applications.* Technischer Bericht Oxford University, 1988.

[MOTW95] Maraist, John, Martin Odersky, David N. Turner und Philip Wadler: *Call-by-name, call-by-value, call-by-need, and the linear lambda calculus.* In: *11'th International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Lousiana, März – April 1995.

[MOW98] Maraist, John, Martin Odersky und Phil Wadler: *The call-by-need lambda calculus.* Technischer Bericht Karlsruhe, 1998.

[MS99] Moran, Andrew und David Sands: *Improvement in a Lazy Context: An Operational Theory for Call-by-Need.* In: *Proc. POPL'99, the 26. ACM SIGPLAN-SIGACT Symposium on Principles of Propramming Languages*, Seiten 43 – 56. ACM Press, Januar 1999.

[MvD96] Meijer, Erik und Joost van Dijk: *Perl for swine: CGI programming in Haskell.* In: *First Workshop on Functional Programming, Buenos Aires, Argentina*, 1996. `http://www.haskell.org`.

[Pit97] Pitts, A. M.: *Operationally-Based Theories of Program Equivalence.* In: Dybjer, P. und A. M. Pitts (Herausgeber): *Semantics and Logics of Computation*, Publications of the Newton Institute, Seiten 241–298. Cambridge University Press, 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.

[PJMS97] Peyton Jones, Simon L. und André Luís de Medeiros Santos: *A transformation-based optimizer for Haskell.* Technischer Bericht Departement of Computing Science, University of Glasgow, Oktober 1997.

[SSH00] Schmidt-Schauss, Manfred und Michael Huber: *A Lambda-Calculus with letrec, case, constructors and non-determinism.* Juli 2000. Entwurf.

[Wad95] Wadler, P.: *Monads for functional programming.* Lecture Notes in Computer Science, 925:24–??, 1995.

[Wad98a] Wadler, Philip: *Functional Programming: An angry half dozen.* SIGPLAN Notices, 33(2):25 – 30, Februar 1998. Functional programming column.

[Wad98b] Wadler, Philip: *Functional Programming: Why no one uses functional languages.* SIGPLAN Notices, 33(8):23 – 27, August 1998. Functional programming column.

[Wad90] Wadler, Philip: *Comprehending Monads.* In: *Conference on Lisp and Functional programming*, Seiten 61–78, Juni 90.

Index