Automatische Laufzeitoptimierung von funktionalen Programmen in einer Kernsprache von Haskell

Diplomarbeit im Studiengang Diplom-BioInformatik
des Fachbereichs Biowissenschaften und
des Fachbereichs Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
Frankfurt am Main

eingereicht von Philipp Eidam
im September 2015

angefertigt in der Professur für Künstliche Intelligenz und Softwaretechnologie
unter Betreuung von Prof. Dr. Manfred Schmidt-Schauß
und PD Dr. David Sabel

Danksagungen

Mein Dank geht an Prof. Manfred Schmidt-Schauß für die Wahl eines Diplomarbeitsthemas, das mir Spaß gemacht und mich gefordert hat, sowie die stets gut gelaunte Betreuung meiner Diplomarbeit.

Mein *besonderer* Dank geht an PD Dr. David Sabel, dessen Sach- und Programmierkenntnisse für das Gelingen dieser Arbeit unverzichtbar gewesen sind und der all meinen Fragen und Problemen mit bewundernswerter Geduld und Hilfsbereitschaft begegnet ist.

Mein Dank geht an Prof. Georg Schnitger für die Bereitschaft kurzfristig als Zweitgutachter für meine Diplomarbeit zu fungieren.

Mein Dank geht an die Community rund um Stack Overflow, den Theoretical Computer Science Stack Exchange, den Computer Science Stack Exchange, den Programmers Stack Exchange und den Mathematics Stack Exchange, durch deren Hilfsbereitschaft fast alle erdenklichen Fragen zu Haskell und dem Lambda-Kalkül schon beantwortet waren, noch bevor ich sie überhaupt stellen konnte.

Mein Dank geht an Frau Ursula Feigenbutz aus dem Prüfungsamt der Biowissenschaften, die den gesamten bürokratischen Teil des Studiums so viel leichter und angenehmer gemacht hat.

Mein Dank geht an meine Frau, Sara Freire Simões de Andrade, die mir nicht nur moralisch in dieser stressigen Zeit beistand, sondern auch dafür gesorgt hat, dass ich mir um viele meiner üblichen Aufgaben keine Sorgen machen musste.

Mein Dank geht an meinen Vater, Ulrich Eidam, für seine Geduld und treue Unterstützung während meines gesamten Studiums.

Inhaltsverzeichnis

0 Zusammenfassung

Im Gegensatz zu Programmtransformationen, sind Programmoptimierungen ein noch relativ unerforschtes Feld. Eine Voraussetzung dafür, Programme automatisch optimieren zu können, ist der Vergleich ihres Ressourcenverbrauchs, wie Laufzeit oder Platzbedarf.

Wird ein Programm als ein Lambda-Ausdruck repräsentiert, wie im Fall von Kernsprachen funktionaler Programmiersprachen, ist die Reduktionslänge, d.h. die Anzahl der Reduktionsschritte in der Reduktion des Ausdrucks zu seiner schwachen Kopfnormalform, eine solche Ressource.

In der vorliegenden Diplomarbeit wird ein Programm vorgestellt, das für Ausdrücke des LR-Kalküls, einer Kernsprache der funktionalen Programmiersprache Haskell, die Reduktionslänge berechnen und vergleichen kann.

1.1 Der Lambda-Kalkül

In den Anfängen der Informatik als Wissenschaft war eine zentrale Aufgabe die Beantwortung der Frage, welche Probleme eigentlich algorithmisch gelöst werden können, also die Suche nach einer Theorie der Berechenbarkeit.
Um eine solche Theorie präzise formulieren zu können, wurden im Laufe der Zeit verschiedene formale Modelle vorgeschlagen, mit deren Hilfe man genaue Aussagen darüber machen kann, was es überhaupt für ein Problem heißt, berechenbar zu sein.

Eines der ersten formalen Berechnungsmodelle waren die von Alan Turing erdachten einfachen, abstrakten Maschinen - heute Turing-Maschinen genannt. Sie sind heute in vielen Bereichen der Informatik das vorherrschende Berechnungsmodell.
Etwa zur selben Zeit wurden allerdings auch alternative Berechnungsmodelle entwickelt und vorgeschlagen, wie die Theorie rekursiver Funktionen von Stephen Kleene und der Lambda-Kalkül von Alonzo Church.

Abhängig davon, welches Berechungsmodell man wählt, erhält man unterschiedliche Antworten auf die Frage, was eine Berechnung eigentlich ist.
Im Gegensatz zu Turing-Maschinen, die Berechnungen als eine zeitliche Abfolge von Befehlen für eine einfache Maschine betrachten, modellieren die beiden letztgenannten Modelle Berechnungen als wiederholte Anwendung von Funktionen auf Argumente (die selbst wieder Funktionen sein können).

Diese Tatsache macht eine Entdeckung der frühen theoretischen Informatik um so überraschender, nämlich dass alle bisher bekannten Berechnungsmodelle formal äquivalent sind, d.h. dass man mit keinem von ihnen mehr berechnen kann als mit den anderen. Dies erlaubt es einem, stets das für ein Problem am besten geeignete Modell zu wählen, ohne befürchten zu müssen, dass die Wahl des Modells einen Einfluss auf die Allgemeinheit der Ergebnisse hat.

In der vorliegenden Diplomarbeit spielt der Lambda-Kalkül als Berechnungsmodell eine zentrale Rolle.

Die Grundlage des Lambda-Kalküls bilden die beiden komplementären Ideen,

- der *Anwendung* (oder *Applikation*) von Funktionen auf konkrete Argumente
- und der Bildung von Funktionen durch *Abstraktion* von konkreten Anwendungen.

Für diese beiden Ideen stellt der Lambda-Kalkül eine einfache und elegante Notation bereit.

1.1.1 Ausdrücke

Das Alphabet des Lambda-Kalküls ist recht klein und besteht nur aus rechter und linker Klammer, dem Symbol λ, und einer unendlichen Menge von Variablen.

Mit Hilfe dieses Alphabets kann die Syntax von Lambda-Ausdrücken rekursiv wie folgt definiert werden (aus [2]):

> **Definition 1.1:**
> Die Menge aller *Lambda-Ausdrücke* ist die kleinste Menge, die die folgenden Bedingungen erfüllt.
> 1. Jede Variable ist ein Lambda-Ausdruck.
> 2. Wenn s und t Lambda-Ausdrücke sind, so ist auch (s t) ein Lambda-Ausdruck.
> 3. Wenn x eine Variable und s ein Lambda-Ausdruck ist,
> so ist auch (λx.s) ein Lambda-Ausdruck.

Dabei werden die durch Regel 2 gebildeten Lambda-Ausdrücke auch *Applikationen* und die durch Regel 3 gebildeten Lambda-Ausdrücke auch *Abstraktionen* genannt.

Die Aufgabe des λ in einer Abstraktion, (λx.s), ist es, die Variable x im Lambda-Ausdruck s zu *binden*. Alle Variablen eines Lambda-Ausdrucks, die nicht durch ein λ gebunden werden, sind *freie Variablen*.

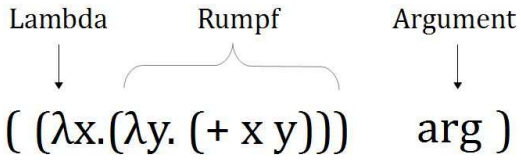
Auf den ersten Blick vielleicht überraschend, reicht diese Syntax auch aus, um mehrstellige Funktionen zu beschreiben. Diese werden nämlich, beim so genannten *Currying*, als eine Folge von ineinander verschachtelten Abstraktionen (also einstelligen Funktionen) betrachtet.
So lässt sich z.B. die zweistellige Funktion (+ x y)[1] durch Abstraktionen darstellen als (λx.(λy. (+ x y))), wobei das Ergebnis der Anwendung dieses Lambda-Ausdrucks auf ein erstes Argument eine einstellige Funktion ist, die zu ihrem jeweiligen Argument den Wert des ersten Arguments hinzuaddiert.
Diese Ketten von Funktionen mit Funktionen als Ausgabe (so genannten *Funktionen höherer Ordnung*) stellen den eigentlichen Trick beim Currying dar.

1.1.2 Reduktion

Um ein vollständiges Kalkül zu erhalten, brauchen wir aber neben den Lambda-Ausdrücken auch noch Regeln, mit denen wir die Ausdrücke bearbeiten können.
Im Kontext des Lambda-Kalküls wird auch davon gesprochen, einen Lambda-Ausdruck zu *reduzieren* und die Regeln dazu heißen dementsprechend *Reduktionsregeln*.

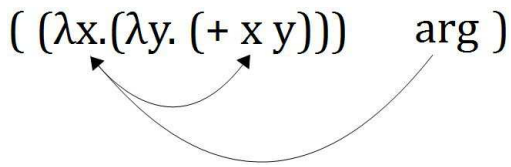
Die wichtigste Reduktionsregel ist die so genannte *β-Reduktion*. Sie ist es nämlich, die eine der beiden Grundideen des Lambda-Kalküls, und zwar die Anwendung einer Funktion auf ein Argument, beschreibt.

Die β-Reduktion kann auf Applikationen angewendet werden, bei denen der erste Unterausdruck eine Abstraktion und der zweite Unterausdruck ein beliebiger Lambda-Ausdruck ist.



1 Das "+" in diesem Beispiel haben wir in unserer Syntax zwar nicht definiert, es dient hier aber auch nur als Abkürzung, um das Beispiel übersichtlicher zu gestalten (für eine mögliche Definition von "+" als Lambda-Ausdruck, siehe den Tests-Abschnitt unter “Komplexe Testfälle”).

Bei der β-Reduktion wird die von dem λ gebundene Variable x im Rumpf der Abstraktion durch das Argument ersetzt,



Als Ergebnis erhält man den Rumpf der Abstraktion, in dem alle Vorkommen der gebundenen Variable durch das Argument ersetzt wurden,

$$(\lambda y.\ (+\ arg\ y))$$

Neben dieser zentralen Regel gibt es (abhängig von der Variante des Lambda-Kalküls) natürlich noch zahlreiche weitere, von denen wir im Abschnitt über die Kernsprache von Haskell noch einige kennenlernen werden.

Hat man einen komplexen Lambda-Ausdruck, kann es passieren, dass man nacheinander mehrere Reduktionsregeln auf ihn anwenden kann, indem man auf das Ergebnis eines jeden Reduktionsschrittes eine weitere Regel anwendet. Auf diese Weise fährt man fort, bis keine weitere Reduktionsregel mehr auf den entstandenen Ausdruck anwendbar ist.
Diesen nicht weiter reduzierbaren Ausdruck nennt man auch die *Normalform* des Anfangs-Ausdrucks.

1.1.3 Reduktionsstrategien

Bei komplexen Lambda-Ausdrücken kann außerdem der Fall eintreten, dass man im selben Reduktionsschritt mehrere Unterausdrücke des Lambda-Ausdrucks mit den jeweils gegebenen Reduktionsregeln reduzieren könnte.

In solchen Fällen muss man entscheiden, in welcher Reihenfolge die verschiedenen Unterausdrücke reduziert werden sollen.

Im folgenden Beispiel ist der zu reduzierende Lambda-Ausdruck eine Applikation, deren erster Unterausdruck eine Abstraktion und deren zweiter Unterausdruck selbst wieder ein reduzierbarer Lambda-Ausdruck ist,

$$((\lambda x.(+ \; x \; x)) \; (+ \; 3 \; 3) \;)$$

Nun könnte man zuerst die β-Reduktion anwenden und das unreduzierte Argument in den Rumpf der Abstraktion einsetzen (d.h. Definitionseinsetzung vor Argumentauswertung),

$$(+ \; (+ \; 3 \; 3) \; (+ \; 3 \; 3))$$

Oder man könnte zunächst das Argument bis zu seiner Normalform reduzieren, bevor man es in den Rumpf der Abstraktion einsetzt (d.h. Argumentauswertung vor Definitionseinsetzung),

$$((\lambda x.(+ \; x \; x)) \; 6 \;)$$

Will man bei der Reduktion eines komplexen Lambda-Ausdrucks nicht für jeden Reduktionsschritt die Reihenfolge der reduzierbaren Unterausdrücke angeben, kann man eine so genannte *Reduktionsstrategie* (oder *Auswertungsstrategie*) festlegen.

Die zwei Hauptstrategien bestehen in der wiederholten Anwendung einer der beiden oben angeführten Möglichkeiten (siehe auch [25]).
Setzt man immer zuerst in Funktionsdefinitionen ein, bevor man die Argumente auswertet, handelt es sich um eine *nicht-strikte* Strategie, die so genannte *Normale Reihenfolge* (oder *Normalordnung*).
Wertet man immer zuerst die Argumente aus, bevor man sie in die Funktionsdefinitionen einsetzt, spricht man von einer *strikten* Strategie, der so genannten *Applikativen Reihenfolge*.

Jede dieser Strategien hat Vor- und Nachteile:

Die Applikative Reihenfolge ist in der Praxis häufig schneller als die Normalordnung, da in vielen Funktionsdefinitionen ein Argument an mehr als einer Stelle eingesetzt wird und die Applikative Reihenfolge ein Argument nur einmal auswertet (unabhängig davon, an wie vielen Stellen es danach eingesetzt wird), während die Normalordnung ein Argument so oft auswerten muss, wie es in der Funktionsdefinition vorkommt[2].

Auf der anderen Seite findet die Applikative Reihenfolge nicht immer die Normalform eines Lambda-Ausdrucks (auch wenn dieser eine hat), wohingegen die Normalordnung immer die Normalform eines Ausdrucks findet, falls diese existiert und der Ausdruck nicht unendlich oft reduziert werden kann (siehe [8], Abschnitt 13.2).
Hat eine Strategie diese Eigenschaft, spricht man auch von einer *normalisierenden Strategie*. Die Normalordnung ist also eine normalisierende Strategie.

Da es uns im Rahmen dieser Diplomarbeit wichtig ist, dass unsere Reduktionsstrategie immer eine (existierende) Normalform für einen gegebenen Lambda-Ausdruck findet, wählen wir eine Variante der Normalordnung als Auswertungsstrategie.

2 Dies gilt für die Normalordnung ohne *Sharing*. Für eine Lösung dieses Problems, siehe die Call-by-need-Strategie im nächsten Abschnitt (unter *Warum als Grundlage?*).

1.2 Funktionale Programmierung und Kernsprachen

Abgesehen von dem Interesse der theoretischen Informatik am Lambda-Kalkül, diente dieser außerdem als Inspiration für ein Programmierparadigma und als Grundlage zahlreicher Programmiersprachen.

1.2.1 Warum *als Inspiration*?

Die beiden Grundideen des Lambda-Kalküls, nämlich die Bildung (Definition) von Funktionen und die Anwendung dieser Funktionen auf konkrete Argumente, waren die Inspiration für das Programmierparadigma der *Funktionalen Programmierung*.

Ein Progammm in der Funktionalen Programmierung besteht nicht aus einer Abfolge von Befehlen (wie z.B. in der imperativen Programmierung), sondern aus einer oder mehreren Funktionsdefinitionen (daher der Name). Ein solches Programm wird dementsprechend aufgerufen, indem die definierte Funktion auf ein konkretes Argument angewendet wird.
In dieser kurzen Beschreibung des Paradigmas kann man bereits deutlich die Nähe zum Lambda-Kalkül erkennen.

Natürlich bringt die Funktionale Programmierung noch andere Eigenheiten mit sich (siehe auch [33]):

In funktionalen Programmiersprachen wird während eines Programmaufrufs die Umgebung des Programms nicht verändert (da es keine Zustandsvariablen für die Umgebung gibt, denen ein Wert zugewiesen werden könnte).
Dies hat zur Folge, dass die Auswertung derselben Funktion mit demselben Argument immer dasselbe Ergebnis liefert. Somit sind unerwartete Nebenwirkungen (so genannte *Seiteneffekte*) ausgeschlossen, die in imperativen Programmen auftreten können, wenn die Zuweisung eines Wertes zu einer Zustandsvariable die Umgebung des laufenden Programms unbeabsichtigterweise verändert.

Aus der Unmöglichkeit der Wertzuweisung zu Zustandsvariablen folgt auch, dass es egal ist, wann ein bestimmter Funktionsaufruf während der Berechung ausgeführt wird (vorausgesetzt, alle benötigten Argumente liegen vor), da die Funktionsauswertungen nicht von einer Umgebung abhängt, die deren Ergebnis beeinflußen könnte. Diese Tatsache macht Funktionale Programmierung auch interessant für Anwendungen, die mit zeitgleichen Prozessen arbeiten (so genannter *Concurrency*).

Abgesehen davon, kommen Programmiertechniken wie *Rekursion* (wie z.B. in rekursiven Funktionsdefinitionen oder rekursiv definierten Daten), Funktionen höherer Ordnung (d.h. Funktionen, die Funktionen als Ausgabe haben) und die Wahl unterschiedlicher Auswertungsstrategien (wie wir bereits angesprochen haben) häufig in der Funktionalen Programmierung zum Einsatz.

1.2.2 Warum *als Grundlage*?

Da der Lambda-Kalkül Turing-vollständig ist, kann man ihn als eine einfache Programmiersprache betrachten, in der man alle berechenbaren Funktion durch Lambda-Ausdrücke repräsentieren kann.

Der Vorteil einer solchen Programmiersprache ist, dass sie eine kleine Syntax und damit eine kleine Anzahl von Konstrukten besitzt, und sich daher leicht implementieren lässt und die Nachweise bestimmter Programmeigenschaften vereinfacht.

Um diese Vorteile zu nutzen, werden viele funktionale Programmiersprachen (wie z.B. Lisp, Scheme, ML oder Haskell) während des Compilierens in Ausdrücke des Lambda-Kalküls übersetzt. Man bezeichnet den Lambda-Kalkül und seine verschiedenen Varianten daher auch als *Kernsprachen* dieser so genannten *höheren Programmiersprachen* (siehe [26]).

Was wir bisher kennengelernt haben, war der einfache Lambda-Kalkül. Der einfache Lambda-Kalkül kann die Übersetzung von funktionalen Programmen in Lambda-Ausdrücke jedoch, je nach Art und Anzahl der Konstrukte in der höheren Programmiersprache, schwierig und unübersichtlich machen.
Daher gibt es verschiedene Möglichkeiten der Erweiterung des einfachen Lambda-Kalküls, die die Übersetzbarkeit und Verständlichkeit verbessern sollen.

Die erste Möglichkeit besteht darin, den einfachen Lambda-Kalkül um *Typen* zu erweitern. Typen geben jeweils die Menge von Werten an, aus der Argumente oder Ergebnisse einer Funktion kommen dürfen. Mit einem getypten Lambda-Kalkül kann man daher Daten mit verschiedenen Typen voneinander unterscheiden und den Wertebereich und den Bildbereich einer Funktion einschränken.

Eine zweite Möglichkeit ist die Wahl unterschiedlicher Auswertungsstrategien (wie wir bereits beim einfachen Lambda-Kalkül gesehen haben).

Die Applikative Reihenfolge (als Vertreter der *strikten* Strategien) und die Normalordnung (als Vertreter der *nicht-strikten* Strategien) wurden bereits im letzten Abschnitt vorgestellt.

Hier soll noch eine effizientere Variante der Normalordnung erwähnt werden, die eine weitere Option darstellt. Wie angesprochen, ist es ein Nachteil der Normalordnung gegenüber der Applikativen Reihenfolge, dass die Normalordnung manchmal das gleiche Argument mehrfach auswerten muss. Erweitert man die Normalordnung um einen Mechanismus, der es ihr erlaubt, bereits ausgewertete Argumente zu speichern, kann sie die Auswertung wiederauftretender Argumente vermeiden und wird dadurch deutlich effizienter. Diese Art der Normalordnung mit Buchführung nennt sich auch *Call-by-need*.

Eine dritte Möglichkeit ist die Erweiterung der Syntax des einfachen Lambda-Kalküls um Konstrukte, die denen höherer Programmiersprachen entsprechen.

Drei solcher Konstrukte, die für die hier verwendete Kernsprache von Haskell wichtig sind, sind die folgenden:

`let/letrec` Mit *let-Ausdrücken* kann man in so genannten *Bindungen*, Variablennamen lokal (in der jeweiligen Funktionsdefinition) bestimmte Ausdrücke zuweisen, die ausgewertet werden und anschließend an den Stellen im Rumpf der Funktion eingesetzt werden, an denen die zugehörigen Namen stehen.

Auch die Erweiterung um *letrec-Ausdrücke* ist möglich, die sich von let-Ausdrücken nur insofern unterscheiden, als dass sie es auch erlauben, die bereits gebundenen Variablennamen in den Bindungen selbst zu verwenden.

`case` Hierbei handelt es sich um eine vereinfachende Notation, um verschachtelte If-then-else leichter und übersichtlicher aufschreiben zu können (dazu muss die Syntax allerdings auch um *Datenkonstruktoren* und *Konstruktor-Applikationen* erweitert werden).

`seq` Der *seq-Operator* ist nützlich, um in Programmiersprachen mit nicht-strikter Auswertungsstrategie (wie z.B. Haskell) bei Bedarf eine strikte Auswertung von Unterausdrücken erzwingen zu können.

1.3 Eine Kernsprache von Haskell

Die Programmiersprache Haskell ist eine rein funktionale Programmiersprache und baut damit auf den Ideen auf, die wir im vorhergehenden Abschnitt kennengelernt haben (siehe auch [15]).

So ist Haskell eine Programmiersprache, die die angesprochene Call-by-need-Auswertungsstrategie verwendet. Die Haskell-spezifische Implementierung der Call-by-need-Strategie nennt man passenderweise auch *lazy evaluation*, da sie Unterausdrücke nur auswertet, wenn sie diese zur Auswertung des Gesamtausdrucks benötigt. Dadurch kann bei der Auswertung von Programm-Ausdrücken viel Zeit eingespart werden und es erlaubt die Verwendung unendlicher Datenstrukturen (z.B. unendlicher Listen), da diese nur soweit ausgewertet werden, wie jeweils gebraucht.

Außerdem ist Haskell eine *statisch getypte* Programmiersprache mit einem sehr mächtigen *Typ-System*. Durch Typ-Systeme können in Programmen Fehler gefunden werden, die darauf beruhen, dass man Operationen auf Daten anwenden möchte, für die sie nicht definiert sind. Eine Möglichkeit, solche Fehler zu vermeiden, ist es, die Datentypen aller Daten und Operationen explizit anzugeben, so dass der Compiler kontrollieren kann, ob alle Datentypen korrekt aufeinander abgestimmt sind. Haskell verlangt vom Programmierer allerdings nicht, all diese Information im Programm zu vermerken. Zu diesem Zweck hat Haskell nämlich so genannte *Typ-Inferenz*. Damit kann der Compiler automatisch die Datentypen der verwendeten Operationen und Argumente ableiten und auf diese Weise viele Inkompatibilitätsfehler finden, bevor das Programm ausgeführt wird.

Aufgrund der Tatsache, dass Programme in Haskell reine Funktionen sind, können mit einfachem *equational reasoning* außerdem Programmtransformation betrieben, Programme direkt von Spezifikationen abgleitet oder Eigenschaften von Programmen nachgewiesen werden.

1.3.1 Der LR-Kalkül

Alle Implementierungen von Haskell benutzen einen erweiterten Lambda-Kalkül mit Call-by-need-Strategie, um dessen Kernsprache zu modellieren.

Wir verwenden hier ebenfalls einen solchen Lambda-Kalkül, der um Letrec-Ausdrücke, Case-Ausdrücke (und Datenkonstruktoren), sowie den Seq-Operator erweitert wurde – auch LR-Kalkül genannt (übernommen aus [27]).

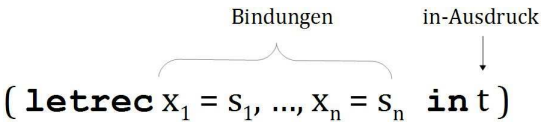
1.3.1.1 Die Syntax

Zunächst geben wir eine kompakte Syntax der LR-Ausdrücke an. Diese lautet:

Variablen:
$x, x_i \in \mathit{Var}$, wobei *Var* die Menge aller Variabeln darstellt

Ausdrücke:
$s, t \in \mathit{Expr} := x \mid (\lambda x.s) \mid (s\ t) \mid (\mathbf{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \mathbf{in}\ t) \mid (\mathbf{seq}\ s\ t) \mid$
$(\ \mathbf{case}_K\ s\ \mathbf{of}\ ((c_{K,1}\ x_1 \ldots x_{ar(c_{K,1})}) \to t_1) \ldots ((c_{K,|D_K|}\ x_1 \ldots x_{ar(c_{K,|D_K|})}) \to t_{|D_K|}))$

Zusätzlich zu Variablen, x, Abstraktionen, (λx.s), und Applikationen, (s t), erlaubt die vorliegende Syntax folgende Konstrukte:



Letrec-Ausdrücke enthalten zwei variable Elemente, die so genannten *Bindungen*, $x_1 = s_1, \ldots x_n = s_n$, und einen LR-Ausdruck t, der auch *in-Ausdruck* genannt wird.
Die Bindungen werden auch L*etrec-Umgebung* genannt und manchmal mit *Env* (von engl. *Environment*) abgekürzt, wenn die genaue Syntax der Bindungen unwichtig ist. Auch Teile der Umgebung können mit *Env* abgekürzt werden,
wie z.B. in (**letrec** Env_1, Env_2 **in** t).

Alle Bindungsvariablen, $x_1, \ldots, x_n$, müssen paarweise disjunkt sein und der Skopus einer Variable x_i umfasst alle s_i und t. Um eine Kette aus Variable-zu-Variable-Bindungen, $x_j = x_{j-1}, x_{j+1} = x_j, \ldots, x_m = x_{m-1}$, darzustellen, benutzen wir auch die Abkürzung $\{ x_i = x_{i-1} \}_{i=j}^{m}$.

$$(\textbf{seq}\ s\ t)$$

Seq-Ausdrücke können, wie gesagt, dazu benutzt werden, in Programmiersprachen mit nicht-strikter Auswertungsstrategie (wie Haskell), die strikte Auswertung von Unterausdrücken zu erzwingen, da in Seq-Ausdrücken, (**seq** s t), der Ausdruck s erfolgreich ausgewertet worden sein muss, bevor der Ausdruck t ausgewertet wird.

$$(\ \textbf{case}_K\ s\ \textbf{of}\ \overbrace{((c_{K,1}\ x_1 \ldots x_{ar(c_{K,1})}) \rightarrow t_1)}^{\text{case-Alternative}} \ldots ((c_{K,|D_K|}\ x_1 \ldots x_{ar(c_{K,|D_K|})}) \rightarrow t_{|D_K|}))$$

Im Fall von Case-Ausdrücken setzen wir zunächst eine feste Menge von *Typkonstruktoren* K und eine endliche, nicht-leere Menge von *Datenkonstruktoren* $D_K = \{ c_{K,1}, \ldots, c_{K,|D_K|} \}$ voraus, wobei jeder Typ-Konstruktor $K \in \mathcal{K}$ eine Stelligkeit $ar(K) \geq 0$ und jeder Datenkonstruktor $c_{K,i} \in D_K$ eine Stelligkeit $ar(c_{K,i}) \geq 0$ hat.

Es gibt ein $case_K$ für jeden vorhandenen Typkonstruktor $K \in \mathcal{K}$ und jeder Case-Ausdruck hat genau eine case-Alternative für jeden Datenkonstruktor $c_{K,i} \in D_K$. Als abkürzende Schreibweise verwenden wir manchmal das Symbol *alts* für die case-Alternativen und schreiben Case-Ausdrücke daher manchmal als ($\textbf{case}_K$ s **of** *alts*).

Die Variablen $x_1, \ldots, x_{ar(c_{K,i})}$ innerhalb der case-Alternative müssen paarweise disjunkt sein und der Skopus der Variablen $x_1, \ldots, x_{ar(c_{K,i})}$ ist der Ausdruck t_i.

$$(c_{K,i}\ s_1 \ \ldots\ s_{ar(c_{K,i})}) \text{ oder } (c\ \vec{s}\ )$$

Außerdem lässt die Syntax die Bildung so genannter *Konstruktor-Applikationen* zu, bei denen ein Datenkonstruktor vollständig mit LR-Ausdrücken gesättigt wird.

1.3.1.2 Die Auswertungsstrategie

Die Call-by-need-Auswertungsstrategie des LR-Kalküls ist in unserem Modell in zwei Schritte aufgeteilt (siehe [27]):
Im ersten Schritt läuft ein Labeling-Algorithmus über den Ausdruck, um denjenigen Unterausdruck (den so genannten *Redex* – für engl. *Reducible Expression*) zu finden, den die Normalordnung im nächsten Schritt reduzieren soll.
Im zweiten Schritt findet dann die eigentliche Normalordnungsreduktion statt, indem auf den gefundenen Unterausdruck eine der Reduktionsregeln angewendet wird.

1.3.1.2.1 Der Labeling-Algorithmus

Die Aufgabe des Labeling-Algorithmus ist es also, denjenigen Unterausdruck eines LR-Ausdrucks zu finden, der als nächstes reduziert werden soll.
Zu diesem Zweck hat der Algorithmus vier Label (*top*, *sub*, *vis*, *nontarg*) zur Verfügung, die er entsprechend der unten aufgeführten Regeln verteilt und ändert.
Das Label *top* dient dazu, die Reduktion des Gesamtausdrucks zu bezeichnen und ist damit auch das Start-Label. Das Label *sub* bezeichnet die Reduktion eines Unterausdrucks, das Label *vis* markiert einen bereits besuchten Unterausdruck und das Label *nontarg* markiert eine bereits besuchte Variable, die kein Ziel einer *cp*-Reduktion ist (siehe Reduktionsregeln unten).

Der Algorithmus verfährt, indem er zunächst den Eingabe-Ausdruck s mit dem top-Label markiert, s^{top} (wobei keiner der Unterausdrücke von s selbst ein weiteres Label haben darf), und anschließend die folgenden Regeln (ebenfalls aus [27]) so lange auf s^{top} anwendet, bis keine weitere Regel mehr anwendbar ist.

$(s\ t)^{sub \vee top}$ [3] $\rightarrow (s^{sub}\ t)^{vis}$

$(\mathbf{letrec}\ Env\ \mathbf{in}\ s)^{top} \rightarrow (\mathbf{letrec}\ Env\ \mathbf{in}\ s^{sub})^{vis}$

$(\mathbf{letrec}\ x = s, Env\ \mathbf{in}\ C[x^{sub}]) \rightarrow (\mathbf{letrec}\ x = s^{sub}, Env\ \mathbf{in}\ C[x^{vis}])$

$(\mathbf{letrec}\ x = s, y = C[x^{sub}], Env\ \mathbf{in}\ t) \rightarrow (\mathbf{letrec}\ x = s^{sub}, y = C[x^{vis}], Env\ \mathbf{in}\ t)$,
wobei C nicht trivial

$(\mathbf{letrec}\ x = s, y = x^{sub}, Env\ \mathbf{in}\ t) \rightarrow (\mathbf{letrec}\ x = s^{sub}, y = x^{nontarg}, Env\ \mathbf{in}\ t)$

$(\mathbf{seq}\ s\ t)^{sub \vee top} \rightarrow (\mathbf{seq}\ s^{sub}\ t)^{vis}$

$(\mathbf{case}\ s\ \mathbf{of}\ alts)^{sub \vee top} \rightarrow (\mathbf{case}\ s^{sub}\ \mathbf{of}\ alts)^{vis}$

$(\mathbf{letrec}\ x = s^{vis \vee nontarg}, y = C[x^{sub}], Env\ \mathbf{in}\ t) \rightarrow$ Fail

$(\mathbf{letrec}\ x = C[x^{sub}], Env\ \mathbf{in}\ s) \rightarrow$ Fail

Hat der Labeling-Algorithmus Erfolg, so findet er im Eingabe-Ausdruck s einen Unterausdruck (immer der Oberausdruck eines *sub*-markierten Unterausdrucks), auf den die Normalordnungsreduktion eine der Reduktionsregeln anwenden kann.
Ist die Normalordnungsreduktion bereits abgeschlossen, oder keine der obigen Markierungsregeln mehr anwendbar, kann es jedoch auch passieren, dass der Labeling-Algorithmus keinen passenden Unterausdruck mehr findet.

[Zur Illustration der Vorgehensweise des Algorithmus werden wir zwei kurze Beispiele per Hand durchrechnen.]

Der erste Beispiel-Ausdruck ist,

$$(\mathbf{letrec}\ x = ((\lambda y.y)\ (\lambda w.w)), y = x, z = y\ \mathbf{in}\ (z\ \lambda u.u))$$

3 Der Ausdruck $a \vee b$ bedeutet dabei "Label a oder Label b".
Der Algorithmus überschreibt außerdem keine nicht-angezeigten Labels.

Auf diesem Ausdruck geht der Labeling-Algorithmus wie folgt vor:

$$(\textbf{letrec}\ x = ((\lambda y.y)\ (\lambda w.w)),\ y = x,\ z = y\ \textbf{in}\ (z\ \lambda u.u))^{top}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = ((\lambda y.y)\ (\lambda w.w)),\ y = x,\ z = y\ \textbf{in}\ (z\ \lambda u.u)^{sub})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = ((\lambda y.y)\ (\lambda w.w)),\ y = x,\ z = y\ \textbf{in}\ (z^{sub}\ \lambda u.u)^{vis})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = ((\lambda y.y)\ (\lambda w.w)),\ y = x,\ z = y^{sub}\ \textbf{in}\ (z^{vis}\ \lambda u.u)^{vis})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = ((\lambda y.y)\ (\lambda w.w)),\ y = x^{sub},\ z = y^{nontarg}\ \textbf{in}\ (z^{vis}\ \lambda u.u)^{vis})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = ((\lambda y.y)\ (\lambda w.w))^{sub},\ y = x^{nontarg},\ z = y^{nontarg}\ \ \textbf{in}\ (z^{vis}\ \lambda u.u)^{vis})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = ((\lambda y.y)^{sub}\ (\lambda w.w))^{vis},\ y = x^{nontarg},\ z = y^{nontarg}\ \textbf{in}\ (z^{vis}\ \lambda u.u)^{vis})^{vis}$$

Der zu reduzierende Unterausdruck ist der Oberausdruck der *sub*-markierten Abstraktion, $(\lambda y.y)^{sub}$, also $((\lambda y.y)^{sub}\ (\lambda w.w))^{vis}$. Auf diesen wendet die Normalordnung im nächsten Schritt die *lbeta*-Reduktionsregel (siehe Reduktionsregeln unten) an.

Der zweite Beispielausdruck ist,

$$(\textbf{letrec}\ x = \lambda u.u,\ y=x,\ z = (y\ y)\ \textbf{in}\ z)$$

Auf diesem Ausdruck geht der Labeling-Algorithmus wie folgt vor:

$$(\textbf{letrec}\ x = \lambda u.u,\ y=x,\ z = (y\ y)\ \textbf{in}\ z)^{top}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = \lambda u.u,\ y=x,\ z = (y\ y)\ \textbf{in}\ z^{sub})^{vis}$$

$$(\textbf{letrec}\ x = \lambda u.u,\ y=x,\ z = (y\ y)^{sub}\ \textbf{in}\ z^{vis})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = \lambda u.u,\ y=x,\ z = (y^{sub}\ y)^{vis}\ \textbf{in}\ z^{vis})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = \lambda u.u,\ y=x^{sub},\ z = (y^{vis}\ y)^{vis}\ \textbf{in}\ z^{vis})^{vis}$$

$$\downarrow$$

$$(\textbf{letrec}\ x = \lambda u.u^{sub},\ y=x^{nontarg},\ z = (y^{vis}\ y)^{vis}\ \textbf{in}\ z^{vis})^{vis}$$

Auf den so gelabelten Ausdruck kann die Normalordnung im Folgeschritt die *cp-e*-Reduktionsregel (siehe Reduktionsregeln unten) anwenden und die *sub*-markierte Abstraktion, $\lambda u.u^{sub}$, an die Stelle des vis-markierten y in $(y^{vis}\ y)^{vis}$ kopieren.

1.3.1.2.2 Die Normalordnungsreduktion

Hat uns der Labeling-Algorithmus den gelabelten LR-Ausdruck geliefert, wenden wir im nächsten Schritt die Normalordnungsreduktion auf diesen an.

Da die Syntax des LR-Kalküls die Bildung komplexerer Ausdrücke zulässt als die Syntax des einfachen Lambda-Kalküls, benötigen wir dementsprechend auch zusätzliche Reduktionsregeln, um Ausdrücke reduzieren zu können, die die neuen Konstrukte enthalten.

Hier sind also die 17 Reduktionsregeln des LR-Kalküls (übernommen aus [27]):

$$(\text{lbeta}) \quad C[((\lambda x.s)^{\mathsf{sub}}\ r) \to C[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$$

Bei der *lbeta*-Regel handelt es sich um eine *Sharing*-Variante der klassischen β-Reduktion, die wir bereits beim einfachen Lambda-Kalkül kennengelernt haben. Beim Sharing werden bereits ausgewertete Ausdrücke gespeichert, so dass sie nicht erneut ausgewertet werden müssen, sollten sie mehrfach im Gesamtausdruck vorhanden sein.

$$\begin{aligned}
(\text{cp-in}) \quad & (\texttt{letrec}\ x_1 = (\lambda x.s)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[x_m^{\mathsf{vis}}]) \\
& \quad \to (\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[(\lambda x.s)]) \\
(\text{cp-e}) \quad & (\texttt{letrec}\ x_1 = (\lambda x.s)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[x_m^{\mathsf{vis}}]\ \texttt{in}\ r) \\
& \quad \to (\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\lambda x.s)]\ \texttt{in}\ r)
\end{aligned}$$

Die *cp-in*- und *cp-e*-Regeln kopieren Abstraktionen aus den Bindungen von Letrec-Ausdrücken entweder in die in-Ausdrücke des Letrec-Ausdrucks oder an andere Stellen in den Bindungen selbst.

$$\begin{aligned}
(\text{llet-in}) \quad & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^{\mathsf{sub}}) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r) \\
(\text{llet-e}) \quad & (\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^{\mathsf{sub}}\ \texttt{in}\ r) \to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)
\end{aligned}$$

Die *llet-in*- und *llet-e*-Regeln vereinen die Letrec-Umgebungen zweier Letrec-Ausdrücke, von denen sich einer im in-Ausdruck oder in den Bindungen des anderen Letrec-Ausdruckes befindet.

$$\begin{aligned}
(\text{lapp}) \quad & C[((\texttt{letrec}\ Env\ \texttt{in}\ t)^{\mathsf{sub}}\ s)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))] \\
(\text{lcase}) \quad & C[(\texttt{case}_K\ (\texttt{letrec}\ Env\ \texttt{in}\ t)^{\mathsf{sub}}\ alts)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_K\ t\ alts))] \\
(\text{lseq}) \quad & C[(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{sub}}\ t)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))]
\end{aligned}$$

Die *lapp*-, *lcase*- und *lseq*-Regeln entfernen einen Letrec-Ausdruck aus der ersten Position in einer Applikation, einem case-Ausdruck oder einem seq-Ausdruck und verschieben die Applikation, den case-Ausdruck oder den seq-Ausdruck in den in-Ausdruck des Letrec-Ausdrucks.

$$
\begin{array}{ll}
\text{(seq-c)} & C[(\texttt{seq}\ v^{\mathsf{sub}}\ t)] \to C[t] \qquad \text{if } v \text{ is a value} \\
\text{(seq-in)} & (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{seq}\ x_m^{\mathsf{vis}}\ t)]) \\
& \quad \to (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[t]) \\
\text{(seq-e)} & (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\texttt{seq}\ x_m^{\mathsf{vis}}\ t)]\ \texttt{in}\ r) \\
& \quad \to (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[t]\ \texttt{in}\ r)
\end{array}
$$

Die Regeln *seq-c*, *seq-in* und *seq-e* werten einen seq-Ausdruck aus. Dazu muss das erste Argument des seq-Ausdrucks allerdings ein *Value* sein (also eine Abstraktion, (λx.s), oder eine Konstruktor-Applikation, $(c\ \vec{s})$) oder eine Variable, die durch eine Konstruktor-Applikation gebunden wird.

$$
\begin{array}{ll}
\text{(case-c)} & C[(\texttt{case}_K\ (c_i\ \overrightarrow{t})^{\mathsf{sub}}\ \ldots((c_i\ \overrightarrow{y})\texttt{->}t)\ldots)] \to C[(\texttt{letrec}\ \{y_i = t_i\}_{i=1}^{n}\ \texttt{in}\ t)] \quad \text{if } n = ar(c_i) \geq 1 \\
\text{(case-c)} & C[(\texttt{case}_K\ c_i^{\mathsf{sub}}\ \ldots\ (c_i\texttt{->}t)\ldots)] \to C[t] \quad \text{if } ar(c_i) = 0 \\
\text{(case-in)} & \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[\texttt{case}_K\ x_m^{\mathsf{vis}}\ \ldots((c_i\ \overrightarrow{z})\texttt{->}t)\ldots] \\
& \quad \to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^{n}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{n}\ \texttt{in}\ t)] \\
& \text{where } n = ar(c_i) \geq 1 \text{ and } y_i \text{ are fresh variables} \\
\text{(case-in)} & \texttt{letrec}\ x_1 = c_i^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[\texttt{case}_K\ x_m^{\mathsf{vis}}\ \ldots\ (c_i\texttt{->}t)\ldots] \\
& \quad \to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[t] \qquad \text{if } ar(c_i) = 0 \\
\text{(case-e)} & \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[\texttt{case}_K\ x_m^{\mathsf{vis}}\ \ldots((c_i\ \overrightarrow{z})\texttt{->}r_1)\ldots], Env\ \texttt{in}\ r_2 \\
& \quad \to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^{n}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ r_1)], Env\ \texttt{in}\ r_2 \\
& \text{where } n = ar(c_i) \geq 1 \text{ and } y_i \text{ are fresh variables} \\
\text{(case-e)} & \texttt{letrec}\ x_1 = c_i^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[\texttt{case}_K\ x_m^{\mathsf{vis}}\ \ldots\ (c_i\texttt{->}r_1)\ldots], Env\ \texttt{in}\ r_2 \\
& \quad \to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^{m}\ldots, u = C[r_1], Env\ \texttt{in}\ r_2 \qquad \text{if } ar(c_i) = 0
\end{array}
$$

Die Regeln *case-c*, *case-in* und *case-e* werten einen case-Ausdruck aus. Dazu muss das erste Argument des case-Ausdrucks eine Konstruktor-Applikation, $(c\ \vec{s})$, des richtigen Typs sein (oder eine Variable, die durch eine solche gebunden wird).

Nachdem wir nun die Reduktionsregeln kennengelernt haben, können wir die Auswertungsstrategie des LR-Kalküls – die Normalordnungsreduktion in zwei Schritten – definieren als:

Definition 1.2:

Sei t ein LR-Ausdruck. Dann definieren wir einen einzelnen *Normalordnungsreduktionsschritt* t → t' als

1. die Anwendung des Labeling-Algorithmus auf den Ausdruck t,
2. und, falls dieser erfolgreich terminieren sollte, die Anwendung einer der LR-Reduktionsregeln auf den so gelabelten Ausdruck.

Führen wir nicht nur einen Normalordnungsreduktionsschritt auf einem Ausdruck aus, sondern wiederholen wir das Verfahren so lange, bis kein weiterer Normalordnungsreduktionsschritt auf dem Ausdruck mehr durchgeführt werden kann, so erreichen wir, ähnlich wie beim einfachen Lambda-Kalkül, eine Normalform des Eingabe-Ausdrucks.

[Um genauer zu sehen, wie die obigen Reduktionsregeln wiederholt auf einen Ausdruck angewendet werden können, siehe die Beispiele für einfache Testfälle im Tests-Abschnitt.]

Da in einem solchen Ausdruck (der nicht weiter vom LR-Kalkül reduziert werden kann) aber immer noch eine oder mehrere Redex vorhanden sein können, spricht man in diesem Fall nicht einfach von *der* Normalform (da eine Normalform üblicherweise gar keine Redex mehr enthält). Man spricht stattdessen von einer *schwachen Kopfnormalform* (auch *WHNF* genannt, für engl. *weak head normal form*).

Definition 1.3:
Eine *schwache Kopfnormalform (WHNF)* ist
1. ein Value v (also eine Abstraktion oder eine Konstruktor-Applikation)
2. oder ein Letrec-Ausdruck, (**`letrec`** *Env* **`in`** v), wobei v ein Value ist,
3. oder ein Letrec-Ausdruck der Form
(**`letrec`** $x_1 = \left(c\ \vec{t}\right), \left\{ x_i = x_{i-1} \right\}_{i=2}^{m}$, *Env* **`in`** x_m).

Zum Beispiel ist der Ausdruck λx.((λy.y)(λz.z)) ist zwar eine WHNF (sie kann vom LR-Kalkül nicht weiter ausgewertet werden), aber keine Normalform im üblichen Sinne (weil ihr Unterausdruck noch weiter ausgewertet werden könnte).

Wie bereits beim einfachen Lambda-Kalkül, kann es passieren, dass unterschiedliche Ausdrücke unterschiedlich oft reduziert werden können.
Daher kann man auch bei LR-Ausdrücken sinnvollerweise von ihrer Reduktionslänge sprechen (d.h. von der Anzahl der Normalordnungsreduktionsschritte, die bis zum Erreichen der WHNF des Ausdrucks ausgeführt werden).

Lässt sich ein Ausdruck zu einer WHNF reduzieren, so spricht man auch davon, dass der Ausdruck *konvergiert*.

Definition 1.4:
Ein Ausdruck t *konvergiert*, geschrieben als t↓, genau dann, wenn es eine WHNF t' gibt, so dass $t \rightarrow t'$, auch geschrieben als t↓t'.

Da der LR-Kalkül jedoch Turing-vollständig ist, kann es bei der Reduktion eines LR-Ausdrucks natürlich auch passieren, dass die Normalordnungsreduktion nicht terminiert und man den Ausdruck nicht zu einer WHNF reduzieren kann. Die Reduktionslänge eines solchen Ausdrucks ist dann unendlich.

Im nächsten Abschnitt werden wir den Begriff der Reduktionslänge wieder aufgreifen und dazu verwenden, ein so genanntes *Improvement* zu definieren.

1.4 Programmtransformation und Improvements

Das Gebiet der Programmtransformation beschäftigt sich mit Operationen, die aus einem Eingabe-Programm ein anderes (Ausgabe-)Programm machen.
Das Konzept der Kernsprache hat uns bereits gezeigt, dass sich Programme auch als Lambda-Ausdrücke darstellen lassen. Daher können wir eine Programmtransformation auch auf Ausdrücken definieren (siehe auch [27]).

> **Definition 1.5:**
> Eine *Programmtransformation P* ist eine binäre Relation auf LR-Ausdrücken.
> Seien s und t zwei LR-Ausdrücke und gilt (s t) ϵ P, dann schreiben wir dafür auch s $\rightarrow^{P}$ t.
> Für eine Menge von Kontexten X und eine Transformation P, ist die Transformation (X, P) der Abschluss von P unter den Kontexten in X,
> d.h. C[s] $\rightarrow^{X,P}$ C[t] genau dann, wenn C ϵ X und s $\rightarrow^{P}$ t.

Ein Kontext ist dabei ein Ausdruck, der an einer Stelle ein "Loch" hat (wir schreiben auch C[.]). Wenn C also ein Kontext ist und s ein Ausdruck, dann ist C[s] der Ausdruck, der entsteht, wenn man den Ausdruck s in das "Loch" des Kontextes eingesetzt hat.

In diesem Sinne können wir die oben angeführten Reduktionsregeln also auch als Programmtransformationen auffassen (bei denen wir das Labeling ignorieren).

Bei einer Programmtransformation kommt es natürlich nicht nur darauf an, ein Programm (oder Ausdruck) in ein beliebiges, anderes Programm (oder Ausdruck) zu verändern. In den meisten Fällen ist es das Ziel, dass das Ausgabe-Programm das gleiche (oder ein erwartet anderes) Programm-Verhalten an den Tag legt wie das Eingabe-Programm, d.h. dass die Programm-Transformation *korrekt* ist.

Um auch diesen Sachverhalt formal fassen zu können, brauchen wir noch einen Begriff dafür, dass zwei Programme das gleiche Programmverhalten zeigen. Man spricht dann auch von der *kontextuellen Äquivalenz* der beiden Programme (oder Ausdrücke).

Definition 1.6:

Seien s und t zwei LR-Ausdrücke. In Bezug auf die operationale Semantik des LR-Kalküls definieren wir dann: Zwei LR-Ausdrücke sind *kontextuell äquivalent*, geschrieben $s \sim_c t$, genau dann, wenn in allen Kontexten C[.] gilt: $C[s]\downarrow \Leftrightarrow C[t]\downarrow$.

Kontextuelle Äquivalenz bedeutet also, dass man ein Programm (oder Ausdruck) durch das andere Programm (oder Ausdruck) ersetzen kann, ohne dass man einen Unterschied im umgebenden Programmkontext bemerken würde.

Damit können wir auch sagen, wann eine Programmtransformation korrekt ist:

Definition 1.7:

Eine Programmtransformation ist *korrekt*, falls sie die kontextuelle Äquivalenz erhält, d.h. falls gilt $P \subseteq \sim_c$.

Da, wie wir oben gesagt haben, die Reduktionsregeln des LR-Kalküls als Programmtransformationen aufgefasst werden können, kann man auch deren Korrektheit beweisen (die Beweise hierfür finden sich in [24]).

In der Forschung wird überwiegend die Analyse und das Beweisen dieser Korrektheits-Eigenschaft von Programmtransformationen betrieben.

Etwas weniger erforscht wird, im Gegensatz dazu, ob die korrekten Programmtransformationen auch *Optimierungen* des Eingabe-Programms darstellen, d.h. ob die Transformationen, bei gleichem Programmverhalten, die Laufzeit oder den Platzbedarf des Programms verringern.

Dafür brauchen wir zunächst wieder Begriffe, anhand derer wir die Laufzeit oder den Platzbedarf eines Programmes (oder Ausdrucks) zumindest annäherungsweise bestimmen können.

Da wir mit LR-Ausdrücken arbeiten werden, werden wir die Ausdrücke anhand der Anzahl der Reduktionsschritte vergleichen, die sie benötigen, um zu ihrer WHNF reduziert zu werden. Zu diesem Zweck werden wir zwei leicht unterschiedliche Maße definieren.

Zum einen `rln`(t), die Reduktionslänge eines Ausdrucks t, als:

Definition 1.8:
Sei t ein geschlossener Ausdruck mit $t \downarrow t_0$.
Dann ist `rln`(t) die Anzahl der *lbeta*-, *case*- und *seq*-Reduktionen in $t \downarrow t_0$.

In der Reduktion eines Ausdrucks zu seiner WHNF werden also nur bestimmte Reduktionsschritte (nämlich *lbeta*-, *case*- und *seq*-Reduktionen) gezählt. Diese Einschränkung lässt sich folgendermaßen begründet (siehe [27], Begründung zu Definition 3.1):

Die cp-Regeln (*cp-e* und *cp-in*) werden nicht gezählt, da jede Anwendung einer dieser Regeln entweder die letzte Regelanwendung überhaupt ist, oder immer von einer lbeta-Reduktion oder einer seq-Reduktion gefolgt wird, die ja in jedem Fall gezählt werden. Daher ist die Anzahl der cp-Reduktionen eines Ausdrucks maximal
(2* `rln`(t)) +1.

Auch alle lll-Regeln (*llet-e*, *llet-in* und *lapp*) werden nicht gezählt, da sie effizienter auf einer abstrakten Maschine implementiert werden können als im Kalkül, und dort alle in einem Schritt ausgewertet werden können, statt Schritt für Schritt durch die Reduktionsregeln des Kalküls, und deshalb vernachlässigbar sind.

Zum anderen `rlnall`(t), die vollständige Reduktionslänge eines Ausdruckes t, als:

Definition 1.9:
Sei t ein geschlossener Ausdruck mit $t \downarrow t_0$.
Dann ist `rlnall`(t) die Anzahl aller Reduktionen in $t \downarrow t_0$.

Der Begriff der `rlnall`(t) ist hauptsächlich in den Beweisen zum Verhältnis von `rln`(t) und `rlnall`(t) (in [27]) relevant. Im Rahmen dieser Diplomarbeit ist der Begriff der `rln`(t) der wichtigere, während die `rlnall`(t) hauptsächlich als interessanter Vergleichswert zu `rln`(t) herangezogen wird.

Mit Hilfe dieser Maße lassen sich nun die Programmtransformationen auf diejenigen einschränken, die nicht nur die kontextuelle Äquivalenz von Eingabe- und Ausgabe-Programm erhalten, sondern zusätzlich fordern, dass das Ausgabe-Programm nicht mehr Laufzeit (oder Platzbedarf) benötigen darf als das Eingabe-Programm.
Entsprechend heißen diese Programmtransformationen dann auch Improvements:

> **Definition 1.10:**
> Seien s und t zwei LR-Ausdrücke. Dann gilt s *verbessert* t, geschrieben $s \lhd t$, genau dann, wenn $s \sim_c t$ und für alle Kontexte C[.], so dass C[s] und C[t] geschlossen sind (d.h. keine freien Variablen mehr enthalten), gilt:
> `rln`(C[s]) ≤ `rln`(C[t]). Wir schreiben dann auch $t \rhd s$.
>
> Eine Programm-Transformation ist ein *Improvement* genau dann,
> wenn gilt $P \subseteq \rhd$.

Ein Improvement liegt also vor, wenn man von zwei kontextuell äquivalenten Ausdrücken, s und t, den Ausdruck t durch s (im Kontext eines größeren Programm-Ausdrucks) ersetzen kann, ohne dass man mehr Berechnungsschritte benötigt, um den gesamten Programm-Ausdruck auszuwerten.

Damit haben wir jetzt den Rahmen abgesteckt, in den sich der Beitrag dieser Diplomarbeit einordnen lässt.

2.1 Ziel

Als einen ersten Schritt auf dem Weg zu einer automatischen Programmoptimierung (durch Testen auf Improvements) brauchen wir also ein Programm, das die `rln`(t) für LR-Ausdrücke berechnen kann.

Dazu muss das Programm,

- einen einzelnen Reduktionsschritt mit Hilfe der Reduktionsregeln durchführen können,
- die so implementierten Einzelschritte so oft auf einen Eingabe-Ausdruck anwenden, bis dieser in seiner WHNF vorliegt (falls diese existiert) und dabei die *lbeta*-, *case*- und *seq*-Reduktionen zählen, um uns anschließend deren Gesamtzahl auszugeben.

Das Ziel dieser Diplomarbeit ist die Implementierung eines solchen Programms.

Wie oben angedeutet, lässt sich das Programm in zwei Unterfunktionen unterteilen, die wir nacheinander definieren werden.

Die erste Unterfunktion wird einen einzelnen Reduktionsschritt auf einem Eingabe-Ausdruck durchführen, also eine einzelne Reduktionsregel anwenden (dies wird die *reduziere*-Funktion sein).
Im Gegensatz zur Aufspaltung der Normalordnungsreduktion in Labeling und eigentlichen Reduktionsschritt (in der Modellierung durch den LR-Kalkül), werden wir in der Implementierung rekursiv im jeweiligen Ausdruck nach dem zu reduzierenden Unterausdruck suchen (analog zum Labeling, aber ohne Labels zu verteilen) und den so gefundenen Unterausdruck anschließend direkt reduzieren.

Die zweite Unterfunktion wird die einzelnen Reduktionsschritte so lange auf den Eingabe-Ausdruck anwenden, bis dieser seine WHNF erreicht hat (falls diese existiert), und dabei die ausgeführten Reduktionsschritte zählen (dies werden die *reduktionsLaenge*- und *reduktionsLaengeVoll*-Funktion sein).

Mit Hilfe dieser beiden Unterfunktionen werden wir anschließend noch zwei weitere, nützliche Funktionen definieren.
Zum einen, eine Funktion, die die `rln`(t) zweier LR-Ausdrücke miteinander vergleicht (dies wird die *hatKleinereRLN*-Funktion sein).
Zum anderen, eine Funktion, die uns die WHNF (sollte diese existieren) eines LR-Ausdrucks ausgibt (dies wird die *normalForm*-Funktion sein).

All diese Funktionen werden in den folgenden Abschnitten ausführlich vorgestellt (das gesamte, zusammenhängende Programm findet sich im Anhang), wobei Anmerkungen das Programmverhalten erklären und Programmierentscheidungen rechtfertigen sollen.

2.1.1 Einschränkungen

In dieser Diplomarbeit werden die oben genannten Funktionen allerdings nicht für die gesamte Kernsprache definiert, sondern auf der folgenden, eingeschränkten Syntax:

Variablen:
$x, x_i \in Var$, wobei *Var* die Menge aller Variabeln darstellt

Ausdrücke:
$s, t \in Expr := x \mid (\lambda x.s) \mid (s\ t) \mid (\mathbf{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \mathbf{in}\ t)$

Neben Variablen, Abstraktionen und Applikationen sind ausschließlich Letrec-Ausdrücke erlaubt.

Durch die eingeschränkte Syntax verringert sich auch die Anzahl der zu implementierenden Reduktionsregeln auf die folgenden (wieder übernommen aus [27]):

$$
\begin{array}{ll}
\text{(lbeta)} & C[((\lambda x.s)^{\mathsf{sub}}\ r) \to C[(\texttt{letrec}\ x = r\ \texttt{in}\ s)] \\
\text{(cp-in)} & (\texttt{letrec}\ x_1 = (\lambda x.s)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[x_m^{\mathsf{vis}}]) \\
 & \quad \to (\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[(\lambda x.s)]) \\
\text{(cp-e)} & (\texttt{letrec}\ x_1 = (\lambda x.s)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[x_m^{\mathsf{vis}}]\ \texttt{in}\ r) \\
 & \quad \to (\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\lambda x.s)]\ \texttt{in}\ r) \\
\text{(llet-in)} & (\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^{\mathsf{sub}}) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r) \\
\text{(llet-e)} & (\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^{\mathsf{sub}}\ \texttt{in}\ r) \to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r) \\
\text{(lapp)} & C[((\texttt{letrec}\ Env\ \texttt{in}\ t)^{\mathsf{sub}}\ s)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))]
\end{array}
$$

Da alle folgenden Funktionen von der korrekten Anwendung der Reduktionsregeln abhängen und darauf aufbauen, ist die Implementierung dieser Regeln der erste Schritt auf dem Weg zu unserem Zielprogramm.

2.2 Der Datentyp

Da unsere eingeschränkte Syntax nur vier Arten von LR-Ausdrücken zulässt, nämlich Variablen, Abstraktionen, Applikationen und Letrec-Ausdrücke, würden die folgenden zwei Datentyp-Deklarationen ausreichen:

```
data Expr v =
   Var v
 | Lam v (Expr v)
 | App (Expr v) (Expr v)
 | Letrec [Binding v] (Expr v)
 deriving(Eq, Show)
```

Im Falle eines Letrec-Ausdrucks muss zusätzlich definiert sein, wie die Bindungen auszusehen haben.

```
data Binding v =
  v :=: (Expr v)
 deriving(Eq, Show)
```

Allerdings werden wir in den folgenden Funktionen bereits den vollständigen Datentyp für LR-Ausdrücke verwenden, um die zukünftige Erweiterung des Programms auf die vollständige Syntax des LR-Kalküls zu vereinfachen.

Der Datentyp für die vollständige Syntax lautet:

```
data Expr label cname v =
   Var v
 | Lam v (Expr label cname v)
 | App (Expr label cname v) (Expr label cname v)
 | Letrec [Binding label cname v] (Expr label cname v)
 | Seq (Expr label cname v) (Expr label cname v)
 | Cons cname [Expr label cname v]
 | Case (Expr label cname v) [CAlt label cname v]
 | Label label (Expr label cname v)
  deriving(Eq, Show)

data Binding label cname v =
  v :=: (Expr label cname v)
deriving(Eq, Show)
```

Dabei bezeichnet label ein beliebiges Label, mit dem man einen LR-Ausdruck durch Einbettung in einen Label-Ausdruck markieren kann. cname steht für einen Datenkonstruktornamen in einem Cons-Ausdruck (einer Konstruktor-Applikation) und in den Case-Alternativen eines Case-Ausdrucks.

Für die Case-Alternativen muss außerdem noch ein zusätzlicher Datentyp definiert werden.

```
data CAlt label cname v =
 CAlt cname [v] (Expr label cname v)
 deriving(Eq, Show)
```

Da wir unseren Expr-Datentyp im Folgenden häufig in Verbindung mit dem Maybe-Datentyp verwenden werden, stellen wir dessen Definition an dieser Stelle auch noch einmal kurz vor.

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

Mit dem Maybe-Datentyp hat man die Möglichkeit, neben unseren LR-Ausdrücken (mit vorangestelltem `Just`), bei Bedarf auch `Nothing` ausgeben zu können, etwa für Fälle, in denen ein LR-Ausdruck nicht weiter reduziert werden kann (in der *reduziere*-Funktion), keine freie Variable gefunden werden kann (in der *findeVar*-Funktion) oder eine bestimmte Letrec-Bindung nicht existiert (in der *exprVonBind*-Funktion), und man diese Tatsache kenntlich machen will.

2.3 Die *reduziere*-Funktion

Die *reduziere*-Funktion ist das Herzstück des Programms. Sie stellt die eigentliche Implementierung der sechs Reduktionsregeln (*lapp*, *lbeta*, *llet-in*, *llet-e, cp-in*, und *cp-e*) dar und führt auf einem gegebenen Ausdruck einen einzelnen Reduktionsschritt anhand dieser Regeln durch.

```
reduziere :: (Eq v, Eq cname, Eq label) =>
[v] -> Maybe (Expr label cname v)
-> (Maybe (Expr label cname v, Bool), [v])
```

Die Funktion erhält zwei Eingaben,
zum einen eine Liste mit Variablennamen, die bisher in dem Eingabe-Ausdruck nicht vorkommen (von hier an *FriVarListe*, für *Frische-Variablennamen-Liste*), und die nur dann zum Einsatz kommt, wenn der Eingabe-Ausdruck oder einer seiner Unterausdrücke umbenannt werden muss (durch die *umbenenne*-Funktion),
und zum anderen einen Eingabe-Ausdruck des Typs (Maybe Ausdruck), um später den rekursiven Aufruf der Funktion zu vereinfachen, da die Ausgabe der Funktion ebenfalls einen Ausdruck der Form (Maybe Ausdruck) enthält.

Die Funktion hat eine Ausgabe des Typs

```
(Maybe (Expr label cname v, Bool), [v])
```

wobei der erste Teil des Tupels, (Maybe (Ausdruck, Bool)), sowohl den reduzierten Eingabe-Ausdruck enthält, sowie einen booleschen Wert, der die Anwendung der *lbeta*-Regel durch die *reduziere*-Funktion anzeigen soll und der später in der *reduktionsLaenge*-Funktion verwendet wird, um die *lbeta*-Reduktionsschritte zu zählen.
Durch die Verwendung des Maybe-Datentyps kann der erste Teil des Ausgabe-Tupels allerdings auch `Nothing` lauten, falls der Eingabe-Ausdruck anhand der gegebenen Reduktionsregeln nicht weiter reduziert werden konnte.

Der zweite Teil des Ausgabe-Tupels ist eine FriVarListe, die entweder die unveränderte Eingabe-Liste ist oder eine Restliste, falls während der Reduktion ein Ausdruck oder Unterausdruck umbenannt werden musste und die in der Umbenennung verwendeten Variablennamen aus der Liste entfernt wurden.

```
reduziere frischeVars (Just (Var x))   = (Nothing, frischeVars)
reduziere frischeVars (Just (Lam x e)) = (Nothing, frischeVars)
```

Erhält die Funktion eine Variable oder eine Abstraktion als Eingabe, so kann sie den erhaltenen Ausdruck nicht weiter reduzieren, und macht an dieser Stelle von dem Maybe-Datentyp Gebrauch und gibt `Nothing` aus (im Sinne von *es existiert keine reduzierte Form dieses Ausdrucks*) zusammen mit der unveränderten FriVarListe (da keine Umbenennung nötig war).

Ist die Eingabe eine Applikation, können, abhängig von der Form des ersten Unterausdrucks der Applikation, drei verschiedene Fälle eintreten:

```
reduziere frischeVars (Just (App (Letrec b e) f))
= (Just (Letrec b (App e f), False), frischeVars)
```

Ist der erste Unterausdruck ein Letrec-Ausdruck, so kann die *lapp*-Reduktionsregel angewendet werden, indem der Letrec-Ausdruck inklusive seiner Bindungsliste aus der Position des ersten Unterausdrucks der Applikation entfernt wird und die Applikation mit dem alten in-Ausdruck und ihrem zweiten Unterausdruck zusammen in den neuen in-Ausdruck des Letrec-Ausdrucks verschoben wird.
Die Anwendung der *lapp*-Regel wird allerdings nicht zur `rln` der Normalordnungsreduktion hinzugezählt, weshalb der boolescher Wert der Ausgabe auf `False` gesetzt wird (im Sinne von *es handelt sich nicht um eine lbeta-Regelanwendung*).

```
reduziere frischeVars (Just (App (Lam x e) f))
= (Just (Letrec [x :=: f] e, True), frischeVars)
```

Ist der erste Unterausdruck eine Abstraktion, so kann die *lbeta*-Reduktionsregel angewendet werden, indem als Ausgabe-Ausdruck ein Letrec-Ausdruck zurückgegeben wird, dessen Bindung als linken Teil die ursprüngliche Variable der Abstraktion und als rechten Teil den zweiten Unterausdruck der Applikation hat, und dessen in-Ausdruck der ursprüngliche Unterausdruck der Abstraktion ist.
Da die *lbeta*-Regel (auf der eingeschränkten Syntax) die einzige Reduktionsregel ist, deren Anwendung zur `rln` der Normalordnungsreduktion hinzugezählt wird, wird der boolesche Wert der Ausgabe entsprechend auf `True` gesetzt (im Sinne von *es handelt sich um eine lbeta-Regelanwendung*).

```
reduziere frischeVars (Just (App e f))
= let (e1, frischeVars')= (reduziere frischeVars (Just e))
  in case e1 of
          Nothing
          -> (Nothing, frischeVars')
          Just (e2, istNormOrdSchritt)
          -> (Just (App e2 f, istNormOrdSchritt), frischeVars')
```

Wurde der Eingabe-Ausdruck auf keines der bisherigen Pattern *gematcht*, kann der erste Unterausdruck (auf der eingeschränkten Syntax) nur noch eine Variable oder selbst wieder eine Applikation sein.
In diesem Fall wird die *reduziere*-Funktion rekursiv mit dem ersten Unterausdruck der Applikation als Eingabe-Ausdruck aufgerufen.
Lässt sich der erste Unterausdruck nicht weiter reduzieren (wie z.B. im Fall einer Variablen), dann lässt sich auch die gesamte Applikation nicht weiter reduzieren und es wird `Nothing` als Resultat für den ursprünglichen Eingabe-Ausdruck ausgegeben, wobei die FriVarListe aktualisiert wird, falls der erste Unterausdruck während seiner Reduktion umbenannt werden musste.

Lässt sich der erste Unterausdruck weiter reduzieren, dann wird er in der ursprünglichen Applikation durch seine reduzierte Form ersetzt, wobei sowohl die Tatsache, ob bei der Reduktion des ersten Unterausdrucks eine *lbeta*-Reduktion angewendet wurde, im Ausgabe-Tupel zurückgegeben wird, als auch die u.U. aktualisierte FriVarListe.

Auch in dem Fall, dass der Eingabe-Ausdruck ein Letrec-Ausdruck ist, können wieder, je nach Art des in-Ausdrucks, drei verschiedene Fälle eintreten:

```
reduziere frischeVars (Just (Letrec b (Letrec c e)))
= (Just (Letrec (b ++ c) e, False), frischeVars)
```

Ist der in-Ausdruck ein weiterer Letrec-Ausdruck, so kann die *llet-in*-Regel angewendet werden, indem ein neuer Letrec-Ausdruck gebildet wird, dessen Bindungsliste die Konkatenation der Bindungsliste des Eingabe-Ausdrucks und der Bindungsliste des in-Ausdrucks ist und dessen neuer in-Ausdruck der Unterausdruck des alten in-Ausdruckes ist.
Da auch die Anwendung dieser Reduktionsregel nicht zur Länge der Normalordnungsreduktion hinzugezählt wird, wird der boolesche Wert der Ausgabe wieder auf `False` gesetzt.

```
reduziere frischeVars (Just (Letrec b (Lam x e)))
= (Nothing, frischeVars)
```

Ist der in-Ausdruck eine Abstraktion, kann der Eingabe-Ausdruck nicht weiter reduziert werden, weshalb in diesem Fall ein `Nothing` ausgegeben wird.

```
reduziere frischeVars (Just (Letrec b e))
 = case (reduziere frischeVars (Just e)) of
        (Just (e1, istNormOrdSchritt), frischeVars')
        -> (Just (Letrec b e1, istNormOrdSchritt), frischeVars')
        (Nothing, frischeVars')
        -> case (findeVar (Just e)) of
                 Just (x, context)
                -> hilf frischeVars' b e x (\z -> Letrec b (context z))
                 Nothing
                 -> (Nothing, frischeVars')
```

Ist der in-Ausdruck eine Variable oder eine Applikation (die einzig verbleibenden Fälle in unserer eingeschränkten Syntax), so wird der Eingabe-Ausdruck vom verbleibenden Letrec-Pattern *gematcht*.
Dieser Programmteil deckt alle fehlenden Reduktionsregeln ab (*llet-e*, *cp-in* und *cp-e*), bei denen die Bindungen des Eingabe-Letrec durchsucht und, in manchen Fällen, verändert werden müssen.
Zunächst wird der in-Ausdruck reduziert.
Ist der in-Ausdruck eine Applikation, die sich zu etwas anderem als `Nothing` reduzieren lässt, wird der, durch die Reduktion der Applikation erhaltene Ausdruck als neuer in-Ausdruck in das Eingabe-Letrec eingesetzt, der boolesche Wert an die, bei der Reduktion des Unterausdrucks verwendeten Reduktionsregeln angepasst und gegebenenfalls die FriVarListe aktualisiert.
Ist der in-Ausdruck allerdings eine Variable oder eine Applikation, die zu `Nothing` reduziert wird, wird die *findeVar*-Funktion mit dem in-Ausdruck als Argument aufgerufen.

Die *findeVar*-Funktion (siehe auch nächsten Abschnitt) sucht, ob es eine am weitesten links stehende, freie Variable in einem Ausdruck gibt (d.h. eine solche, die nicht durch ein Lambda gebunden wird) und wenn dies der Fall ist, gibt sie diese aus, zusammen mit dem Kontext, in dem diese steht (d.h. den gesamten Eingabe-Ausdruck mit einem "Loch" an der Position der gefundenen Variable).

Findet die *findeVar*-Funktion keine freie Variable, dann kann das Eingabe-Letrec nicht weiter reduziert werden und die *reduziere*-Funktion gibt `Nothing` und die aktualisierte FriVarListe aus.
Kann eine freie Variable gefunden werden, so wird diese Variable mit ihrem Kontext (eingebettet in das Eingabe-Letrec) an die *hilf*-Funktion übergeben, zusammen mit den Bindungen und dem in-Ausdruck des Eingabe-Letrec, sowie der aktualisierten FriVarListe.

Die *hilf*-Funktion (siehe auch übernächsten Abschnitt) sucht in den Bindungen des Eingabe-Letrec nach einer Bindung mit der gefundenen Variable auf der linken Seite. Falls es eine solche Bindung findet, nimmt die *hilf*-Funktion die rechte Seite dieser Bindung (d.h. den dazugehörigen Ausdruck) und je nachdem von welcher Art der so erhaltene Ausdruck ist, verändert sie entweder die Bindungen des Eingabe-Letrec (entsprechend den Regel *llet-e* und *cp-e*) oder dessen in-Ausdruck (entsprechend der Regel *cp-in*), oder gibt `Nothing` zurück.

```
reduziere frischeVars Nothing = (Nothing, frischeVars)
```

Diese Ergänzung zur Definition der *reduziere*-Funktion ist notwendig, damit die *hilf*-Funktion, auch wenn keine Bindung mit der gesuchten Variable in den Bindungen des Eingabe-Letrec existiert (und sie auf eine entsprechende Anfrage den Wert `Nothing` zurückbekommt), trotzdem den *reduziere*-Schritt durchführen kann (siehe Abschnitt über *hilf*-Funktion).

2.4 Die *findeVar*-Funktion

Wie bereits im vorherigen Abschnitt beschrieben, sucht die *findeVar*-Funktion, ob der Eingabe-Ausdruck der *reduziere*-Funktion an der zu reduzierenden Position eine freie Variable enthält. Ist dies der Fall, so gibt die *findeVar*-Funktion die Variable zusammen mit dem Kontext aus, in dem die Variable steht.

```
findeVar :: Maybe (Expr label cname v)
-> Maybe (v, Expr label cname v -> Expr label cname v)
```

Wie schon bei der *reduziere*-Funktion ist die Eingabe der *findeVar*-Funktion eine Instanz des Maybe-Datentyps, also entweder ein LR-Ausdruck (mit vorangestelltem `Just`) oder `Nothing` (siehe letzten Absatz dieses Abschnittes für eine Begründung, warum die Funktion auch `Nothing` als mögliche Eingabe akzeptieren muss).

Die Ausgabe der *findeVar*-Funktion ist entweder `Nothing` oder ein Tupel bestehend aus der gefundenen, freien Variable und einer *context*-Funktion. Die Kontexte als Ausdrücke mit "Loch" haben wir bereits weiter oben kennengelernt. Hier werden Kontexte als Funktionen implementiert, d.h. für einen Kontext C wird eine Abstraktion $\lambda x \rightarrow C[x]$ erzeugt, so dass das Einsetzen von s in C[.] genau dem Anwenden der *context*-Funktion auf den Ausdruck s entspricht.

```
findeVar (Just (Lam x e))     = Nothing
findeVar (Just (Letrec b e)) = Nothing
```

Zu beachten ist, dass die findeVar-Funktion nur aufgerufen wird, wenn der Eingabe-Ausdruck der *reduziere*-Funktion ein Letrec-Ausdruck ist. Folglich werden Eingabe-Ausdrücke mit Abstraktion oder Letrec-Ausdrücken als in-Ausdrücken bereits von den ersten beiden *reduziere*-Letrec-Pattern abgefangen.

Aus einem ähnlichen Grund können Applikationen, die (bei beliebiger Applikations-Schachtelung) eine Abstraktion oder einen Letrec-Ausdruck als ersten Unterausdruck enthalten, keine möglichen Eingaben für die *findeVar*-Funktion sein, da solche Applikationen bereits vorher (im Aufruf der *reduziere*-Funktion im letzten *reduziere*-Letrec-Pattern) durch die *reduziere*-Applikation-Pattern abgefangen, reduziert und vor dem *findeVar*-Funktionsaufruf ausgegeben worden wären.
Es kann sich bei den Eingabe-Ausdrücken für die *findeVar*-Funktion also nur noch um eine Variable oder eine (beliebig geschachtelte) Applikation handeln, die zu **Nothing** reduziert werden kann.
Daher gibt die *findeVar*-Funktion bei einer Abstraktion oder einem Letrec-Ausdruck als Eingabe **Nothing** aus.

```
findeVar (Just (Var y))       = Just (y, id)
```

Ist der Eingabe-Ausdruck eine Variable, handelt es sich um eine freie Variable. Diese wird also direkt ausgegeben, zusammen mit der Identitätsfunktion als ihrer *context*-Funktion, die die Variable einfach durch einen in sie eingegebenen Ausdruck ersetzt.

```
findeVar (Just (App e f))
= case (findeVar (Just e)) of
        Nothing
        -> Nothing
        Just (y, context)
        -> Just (y, \z -> App (context z) f)
```

Ist der Eingabe-Ausdruck eine Applikation, die zu **Nothing** reduziert werden kann, wird die *findeVar*-Funktion (entsprechend unserer Vorgabe der Normalordnung als Auswertungsstrategie) rekursiv mit dem ersten Unterausdruck der Applikation aufgerufen (wie schon in der *reduziere*-Funktion).
Findet sich im ersten Unterausdruck keine freie Variable wird **Nothing** ausgegeben.

Enthält der erste Unterausdruck jedoch eine freie Variable, wird die so gefundene Variable ausgegeben und ihre *context*-Funktion aktualisiert, indem der bisherige Kontext in einen zusätzlichen Applikations-Kontext eingebettet wird (oder in mehrere, für den Fall, dass der Eingabe-Ausdruck mehrere verschachtelte Applikationen enthielt).

```
findeVar Nothing = Nothing
```

Ähnlich wie die *reduziere*-Funktion, muss auch die *findeVar*-Funktion mit einem `Nothing` als Eingabe umgehen können, da die *hilf*-Funktion den *findeVar*-Funktionsaufruf auch ausführen können muss, wenn keine Bindung mit der gesuchten Variable in den Bindungen des Eingabe-Letrec existiert und sie daher auf eine Anfrage nach einer solchen Bindung, ein `Nothing` zurückbekommt (siehe Abschnitt über *hilf*-Funktion).

2.5 Die *hilf*-Funktion

Wie die *findeVa*r-Funktion, wird auch die *hilf*-Funktion nur aufgerufen, wenn der Eingabe-Ausdruck der *reduziere*-Funktion ein Letrec-Ausdruck ist.

Die *hilf*-Funktion durchsucht die Bindungen des Eingabe-Letrec nach einer Bindung mit einer bestimmten Variable auf der linken Seite. Findet die Funktion eine solche Bindung, nimmt sie die dazugehörige rechte Seite der Bindung und verändert, je nach Art der rechten Seite, entweder die Bindungen des Eingabe-Letrec oder dessen in-Ausdruck.

```
hilf :: (Eq v, Eq cname, Eq label) =>
[v] -> [Binding label cname v] -> Expr label cname v ->
v -> (Expr  label cname v -> Expr label cname v)
-> (Maybe (Expr label cname v, Bool), [v])
```

Die *hilf*-Funktion erhält fünf Werte als Eingaben:
Die Funktion bekommt die aktuelle FriVarListe, die Bindungsliste eines Letrec-Ausdrucks, den in-Ausdruck eines Letrec-Ausdrucks, eine Variable und einen Kontext.
Beim ersten *hilf*-Aufruf sind dies neben der aktuellen FriVarListe, die Bindungsliste und der in-Ausdruck (in dem die freie Variable gefunden wurde) des Eingabe-Letrecs der *reduziere*-Funktion, sowie die gefundene, freie Variable und ihr ursprünglicher Kontext, eingebettet in einen Letrec-Kontext.
Als Ausgabe hat die *hilf*-Funktion, ebenso wie die *reduziere*-Funktion, ein Tupel aus LR-Ausdruck, booleschem Wert und aktueller FriVarListe.

```
hilf frischeVars b e x context =
let r = (exprVonBind x b)
```

Die *exprVonBind*-Funktion erhält als Eingabe einen Variablennamen und eine Bindungsliste und durchsucht die Bindungsliste nach einer Bindung, deren linke Seite der eingegebenen Variablen entspricht. Findet die Funktion eine solche Bindung, gibt sie die rechte Seite der Bindung (also den zugehörigen Ausdruck) aus, ansonsten gibt sie `Nothing` aus.

Je nachdem von welcher Form der so gefundene Ausdruck ist, verfährt die *hilf*-Funktion unterschiedlich:

```
in case r of
Just (Letrec b1 e1)
-> let b2 = (aktualisiereBind e1 x b)
   in (Just (Letrec (b1 ++ b2) e, False), frischeVars)
```

Ist der Bindungs-Ausdruck ein Letrec-Ausdruck, so gibt die *hilf*-Funktion wieder einen Letrec-Ausdruck aus, dessen Bindungsliste aber die Konkatenation der Bindungsliste des Bindungs-Ausdrucks und der aktualisierten Eingabe-Bindungsliste ist.

Die *aktualisiereBind*-Funktion erhält als Eingabe einen LR-Ausdruck, einen Variablennamen und eine Bindungsliste. Hat die Funktion die Bindung mit der eingegebenen Variable als linke Seite gefunden, ersetzt sie die rechte Seite dieser Bindung durch den eingegebenen LR-Ausdruck.

In diesem Fall wird die Eingabe-Bindungsliste der *hilf*-Funktion aktualisiert, indem die rechte Seite der Bindung der Eingabe-Variable (also der durch die *exprVonBind*-Funktion gefundene Ausdruck, nämlich das Bindungs-Letrec) durch den in-Ausdruck des Bindungs-Letrec ersetzt wird.
Der in-Ausdruck des Eingabe-Letrec bleibt unverändert.
Zusätzlich beinhaltet die Ausgabe noch den booleschen Wert `False` (da kein zu zählender Reduktionsschritt ausgeführt wurde) und die unveränderte FriVarListe.

Das Ergebnis entspricht also der Anwendung der *llet-e*-Reduktionsregel auf das Eingabe-Letrec der *reduziere*-Funktion.

```
Just (Var z)
-> hilf frischeVars b e z context
```

Ist der Bindungs-Ausdruck eine Variable, so wird die *hilf*-Funktion rekursiv mit der ursprünglichen Bindungsliste, dem ursprünglichen LR-Ausdruck und dem ursprünglichen Kontext (damit ein möglicherweise im erneuten Aufruf gefundener Ausdruck an der Stelle der ursprünglichen Variable ersetzt werden kann), aber mit dem Variablennamen des Bindungs-Ausdrucks aufgerufen.
Dieser Fall implementiert das Auffinden von *Variablen-Ketten*, in denen zwei Variablen über mehrere Variable-zu-Variable-Bindungen miteinander verbunden sind.

```
Just (Lam a e')
-> let (umbenannteAbstr, frischeVars') = umbenenne (Lam a e') frischeVars
   in  (Just ((context umbenannteAbstr), False), frischeVars')
```

Ist der Bindungs-Ausdruck eine Abstraktion, so wird zunächst die *umbenenne*-Funktion mit dieser Abstraktion und der aktuellen FriVarListe aufgerufen.

Die *umbenenne*-Funktion (siehe auch nächsten Abschnitt) hat zwei Listen zur Verfügung, eine Liste bisheriger Belegungen und die Liste unbenutzter Variablennamen. Sie erhält diese beiden Listen und einen umzubenennenden Ausdruck als Eingabe. Anschließend benennt sie alle durch ein Lambda gebundenen oder in Letrec-Bindungen verwendeten Variablen im Eingabe-Ausdruck anhand der Belegungs-Liste (die während der Umbenennung länger werden kann) um. Sind Variablen des Ausdrucks frei, werden sie nicht umbenannt.
Indem die Bindungs-Abstraktion zuerst umbenannt wird, soll verhindert werden, dass durch das spätere Einsetzen der Abstraktion in den Kontext gleichlautende Variablen unabsichtlich gebunden werden, oder dass in einem Letrec-Ausdruck mehrdeutige Bindungen entstehen (wenn dieselbe Variable auf der linken Seite mehrerer Bindungen vorkommt).

Anschließend wird die umbenannte Abstraktion der *context*-Funktion als Eingabe übergeben und der resultierende Ausdruck dient, zusammen mit dem booleschen Wert `False` (da keine *lbeta*-Reduktion durchgeführt wurde) und der aktualisierten FriVarListe als Ausgabe der hilf-Funktion.
Der ausgegebene Ausdruck entspricht der Anwendung der *cp-in*-Reduktionsregel auf das Eingabe-Letrec der *reduziere*-Funktion.

```
otherwise
-> case (reduziere frischeVars r) of
        (Just (r1, istNormOrdSchritt), frischeVars')
         -> let b1 = (aktualisiereBind r1 x b)
            in (Just (Letrec b1 e, istNormOrdSchritt), frischeVars')
```

Ist der Bindungs-Ausdruck eine Applikation oder ein `Nothing` (falls keine passende Bindung existiert), wird der erhaltene Ausdruck zunächst der *reduziere*-Funktion übergeben. An dieser Stelle wird auch ersichtlich, warum die *reduziere*-Funktion auch für Eingaben von `Nothing` definiert sein muss.

Lässt sich der Bindungs-Ausdruck zu etwas anderem als **Nothing** reduzieren (was nur für bestimmte Applikation der Fall sein kann), dann wird die Eingabe-Bindungsliste wieder aktualisiert, indem das Resultat der Reduktion des Bindungs-Ausdrucks auf der rechten Seite der Bindung mit der Eingabe-Variable eingesetzt wird. Der Eingabe-in-Ausdruck des Letrec-Ausdrucks wird beibehalten und der boolesche Wert angepasst (je nachdem, ob bei der Reduktion des Bindungs-Ausdrucks die *lbeta*-Regel angewendet wurde), sowie die FriVarListe u.U. aktualisiert.

```
(Nothing, frischeVars')
-> case (findeVar r) of
        Nothing
        -> (Nothing, frischeVars')
        Just (y, context1)
         -> hilf frischeVars' b e y (\z -> Letrec ((x :=: (context1 z)):
                                                    (löscheBind x b)) e)
```

Lässt sich der Bindungs-Ausdruck nur zu **Nothing** reduzieren, so wird die *findeVar*-Funktion mit ihm als Eingabe aufgerufen.

Findet die *findeVar*-Funktion keine passende, freie Variable, so wird **Nothing** (mit der aktualisierten FriVarListe) ausgegeben.

Findet die findeVar-Funktion eine solche Variable im Bindungs-Ausdruck, so wird rekursiv die *hilf*-Funktion aufgerufen, wobei die FriVarListe, die Bindungen und der LR-Ausdruck als Eingaben unverändert bleiben. Als neue Variable wird die im Bindungs-Ausdruck gefundene Variable an die *hilf*-Funktion übergeben und der Kontext der Variable wird in einen Letrec-Kontext eingebettet, in dessen Bindungen die Bindung mit der alten Eingabe-Variable auf der linken Seite gelöscht und stattdessen eine neue Bindung eingefügt wurde, auf deren linker Seite die alte Eingabe-Variable und auf deren rechter Seite der Kontext der Variable des Bindungs-Ausdrucks steht.

Die *löscheBind*-Funktion erhält eine Variable und eine Bindungsliste als Eingabe und löscht die Bindung aus der Liste, die auf ihrer linken Seite die Eingabe-Variable enthält.

Der ausgegebene Ausdruck entspricht hier der Anwendung der *cp-e*-Reduktionsregel auf das Eingabe-Letrec der *reduziere*-Funktion.

2.6 Die *umbenenne*-Funktion

Die *umbenenne*-Funktion benennt einen eingegebenen Ausdruck mit Hilfe einer Belegungsliste und einer Liste unbenutzter Variablennamen um. Die Funktion sucht alle, durch ein Lambda gebundenen oder in Letrec-Bindungen verwendeten Variablen und kontrolliert anhand der Belegungsliste, ob bereits eine Zuordnung des gefundenen Variablennamen zu einem neuen Variablennamen existiert. Ist dies der Fall, so wird die gefundene Variable der Belegung entsprechend umbenannt, ansonsten behält die Variable ihren Namen.

Zunächst definieren wir uns eine Hilfsfunktion.

```
umbenenneHilf :: (Eq v) =>
[(v, v)] -> Expr label cname v -> [v]
-> (Expr label cname v, [v])
```

Die *umbenenneHilf*-Funktion erhält drei Werte als Eingabe:
Zum einen, eine Liste der bisherigen Belegungen (definiert als ein Tupel zweier Variablennamen), in der vermerkt ist, welche Variablen bereits auf welche Weise umbenannt wurden, zum anderen, einen LR-Ausdruck, der umbenannt werden soll, und außerdem die bereits vielfach erwähnte FriVarListe, die in dieser Funktion endlich zum Einsatz kommt.

Die Ausgabe der Funktion ist der umbenannte Ausdruck und die aktualisiert Liste noch unbenutzter Variablennamen.

```
umbenenneHilf belegungs (Var x) frischeVars
= case lookup x belegungs  of
        Nothing
        -> (Var x, frischeVars)
        Just y
        -> (Var y, frischeVars)
```

Im einfachsten Fall ist der Eingabe-Ausdruck eine Variable. Dann sucht die *umbenenneHilf*-Funktion in der Belegungsliste eine Belegung, deren linke Seite der Eingabe-Variable entspricht.
Findet die Funktion eine solche Belegung, wird der Eingabe-Ausdruck (d.h. die Variable) einfach, entsprechend der Belegung, umbenannt.
Gibt es keine passende Belegung, wird die nicht-umbenannte Eingabe-Variable wieder ausgegeben, da es sich dann um eine Variable handelt, die nicht durch ein Lambda oder in einer Letrec-Bindung gebunden ist.

```
umbenenneHilf belegungs (Lam x e) (f : frischeVars)
= let (e', frischeVars') = umbenenneHilf ((x, f) : belegungs) e frischeVars
  in (Lam f e', frischeVars')
```

Ist der Eingabe-Ausdruck eine Abstraktion, so wird zunächst die durch das Lambda gebundene Variable (mit dem ersten noch unbenutzten Variablennamen) umbenannt und die entsprechende Belegung in die Belegungsliste aufgenommen. Anschließend wird der Unterausdruck der Abstraktion mit Hilfe der neuen Belegungsliste und der aktualisierten FriVarListe durch die *umbenenneHilf*-Funktion umbenannt. Die Ausgabe besteht dann aus der Abstraktion mit umbenannter, gebundener Variable und einem entsprechend umbenannten Unterausdruck (neben der aktualisierten FriVarListe).

```
umbenenneHilf belegungs (App e1 e2) frischeVars
= let (e1', frischeVars')  = umbenenneHilf belegungs e1 frischeVars
      (e2', frischeVars'') = umbenenneHilf belegungs e2 frischeVars'
  in (App e1' e2', frischeVars'')
```

Ist der Eingabe-Ausdruck eine Applikation, so wird zunächst deren erster Unterausdruck umbenannt (und dabei u.U. frische Variablennamen verbraucht) und die FriVarListe aktualisiert. Mit dieser aktualisierten Liste wird anschließend der zweite Unterausdruck umbenannt.
Neben der Applikation mit umbenanntem ersten und zweiten Unterausdruck wird außerdem die u.U. zweifach aktualisierte FriVarListe ausgegeben.

```
umbenenneHilf belegungs (Letrec b e) frischeVars
= let bindvars           = map (\(x :=: r) -> x) b
      l                  = length bindvars
      umbenennung        = zip bindvars frischeVars
      frischeVars'       = drop l frischeVars
      belegungs'         = umbenennung ++ belegungs
      (b', frischeVars'')  = umbenenneBinds belegungs' b frischeVars'
      (e', frischeVars''') = umbenenneHilf belegungs' e frischeVars''
  in (Letrec b' e', frischeVars''')
```

Ist der Eingabe-Ausdruck ein Letrec-Ausdruck, wird zunächst eine Liste mit Variablen erstellt, die auf der linken Seite einer Letrec-Bindung vorkommen (bindvars). Anschließend werden aus dieser Liste und der FriVarListe neue Belegungen erstellt, wobei eine Bindungsvariable jeweils einem frischen Variablennamen zugeordnet wird (umbenennung). Die dabei verwendeten Variablennamen werden aus der FriVarListe entfernt (frischeVars'). Die so erzeugten, neuen Belegungen werden zu den bereits existierenden Belegungen hinzugefügt (belegungs') und die Bindungen des Letrec-Ausdrucks mit Hilfe dieser neuen Belegungsliste und der aktualisierten FriVarListe (frischeVars') durch die

umbenenneBinds-Funktion (siehe unten) umbenannt, wobei u.U. weitere Variablennamen verbraucht werden können und die FriVarListe entsprechend erneut aktualisiert werden muss (frischeVars''). Zu guter Letzt, wird der in-Ausdruck des Letrec-Ausdrucks unter Zuhilfenahme der neuen Belegungsliste und der zweifach aktualisierten FriVarListe umbenannt.
Die Ausgabe der *umbenenneHilf*-Funktion besteht anschließend sowohl aus dem Letrec-Ausdruck mit umbenannten Bindungen und umbenanntem in-Ausdruck, als auch aus der dreifach aktualisierten FriVarListe.

```
umbenenneBinds :: (Eq v) =>
[(v, v)] -> [Binding label cname v] -> [v] ->
([Binding label cname v], [v])
```

Analog zur *umbenenneHilf*-Funktion besteht die Eingabe der *umbenenneBinds*-Funktion aus einer Belegungsliste und einer Liste unbenutzter Variablennamen, anstelle des Eingabe-Ausdrucks dient jedoch eine Bindungsliste als zusätzliches Argument.
Die Ausgabe ist, wie zu erwarten, die umbenannte Bindungsliste und die aktualisierte FriVarListe.

```
umbenenneBinds _ [] frischeVars = ([], frischeVars)

umbenenneBinds belegungs ((x :=: e) : xs) frischeVars
= let (e', frischeVars') = umbenenneHilf belegungs e frischeVars
      (xs', frischeVars'') = umbenenneBinds belegungs xs frischeVars'
      x'                   = fromJust (lookup x belegungs)
  in ((x' :=: e') : xs', frischeVars'')
```

Zur Umbenennung der Bindungsliste wird zunächst die erste Bindung der Liste umbenannt und dann rekursiv auf der Liste fortgefahren.

Eine Bindung wird umbenannt, indem die Variable (auf der linken Seite) der Bindung mit Hilfe der bereits vorhandenen Belegungen umbenannt wird. Anschließend wird der Ausdruck (auf der rechten Seite) der Bindung mit Hilfe der Belegungsliste und der FriVarListe umbenannt. Ist dies geschehen, wird der Rest der Bindungsliste rekursiv mit Hilfe der Belegungsliste und der (nach der Umbenennung der ersten rechten Seite) aktualisierten FriVarListe umbenannt.

Die *umbenenne*-Funktion selbst besteht nun aus einem einfachen Aufruf der *umbenenneHilf*-Funktion mit einer leeren Belegungsliste als festem Anfangsargument.

```
umbenenne :: (Eq v) =>
Expr label cname v -> [v] ->
(Expr label cname v, [v])

umbenenne expr vars = umbenenneHilf [] expr var
```

2.7 Die Ziel-Funktionen

Nachdem wir nun die *reduziere*-Funktion implementiert und beschrieben haben, können wir diese verwenden, um die Funktionen zu definieren, die für einen gegebenen LR-Ausdruck berechnen,

- wieviele *lbeta*-Reduktion in seiner Reduktion angewendet wurden (die *reduktionsLaenge*-Funktion),
- wieviele Reduktionsregeln *insgesamt* in seiner Reduktion angewendet wurden (die *reduktionsLaengeVoll*-Funktion),
- ob er im Vergleich zu einem zweiten eingegebenen Ausdruck eine kleinere Reduktionslänge hat (die *hatKleinereRLN*-Funktion),
- und wie seine WHNF aussieht (die *normalForm*-Funktion).

Da sich all diese Funktion relativ einfach und kurz mit der *reduziere*-Funktion definieren lassen, werden diese Funktionen im folgendem Abschnitt zusammen vorgestellt.

2.7.1 Die *reduktionsLaenge*-Funktion

Die *reduktionsLaenge*-Funktion zählt die *lbeta*-Reduktionen innerhalb der Normalordnungsreduktion eines LR-Ausdrucks.

Dazu definieren wir uns zunächst wieder eine Hilfsfunktion, namens *reduktionsLaengeHilf*:

```
reduktionsLaengeHilf :: (Eq v, Eq cname, Eq label) =>
Maybe (Expr label cname v) -> Integer -> [v]
-> Integer
```

Die *reduktionsLaengeHilf*-Funktion erhält als Eingaben den LR-Ausdruck, dessen `rln` berechnet werden soll, die Anzahl der bisher durchgeführten *lbeta*-Reduktionen, und die schon bekannte FriVarListe (für den Fall, dass der Ausdruck während der Reduktion umbenannt werden muss).
Als Ausgabe gibt sie uns die `rln`, also in unserem Fall (auf der eingeschränkten Syntax) die Anzahl der *lbeta*-Regelanwendungen, aus.

```
reduktionsLaengeHilf expr schritte frischeVars
= case reduziere frischeVars expr of
        (Nothing, frischeVars')
        -> schritte
        (Just (expr', istNormOrdSchritt), frischeVars')
        -> if istNormOrdSchritt
           then reduktionsLaengeHilf (Just expr') (schritte + 1) frischeVars'
           else reduktionsLaengeHilf (Just expr') schritte frischeVars'
```

Die *reduktionsLaengeHilf*-Funktion ruft die *reduziere*-Funktion auf dem Eingabe-Ausdruck, zusammen mit der FriVarListe auf.
Für den Fall, dass der Eingabe-Ausdruck in einem einzelnen Schritt der *reduziere*-Funktion zu `Nothing` auswertet, gibt die *reduktionsLaengeHilf*-Funktion die Anzahl der bis dahin ausgeführten lbeta-Reduktionen aus.
Lässt sich der Eingabe-Ausdruck in einem einzelnen Schritt zu einem neuen LR-Ausdruck reduzieren, und die Ausgabe der *reduziere*-Funktion besteht daher aus eben diesem Ausdruck, dem booleschen Wert und der u.U. aktualisierten FriVarListe, so kontrolliert die *reduktionsLaengeHilf*-Funktion, ob der boolesche Wert `True` ist (falls während der Reduktion des Eingabe-Ausdrucks die *lbeta*-Regel angewendet wurde) oder `False` (falls beliebige Regeln, mit Ausnahme der *lbeta*-Regel, während der Reduktion des Eingabe-Ausdrucks angewendet wurden).

Ist der Wert `True`, so wird die *reduktionsLaengeHilf*-Funktion rekursiv mit dem neu entstandenen Ausdruck und der FriVarListe aufgerufen, wobei die Anzahl der Schritte um eins erhöht wird.
Ist der Wert `False`, wird die *reduktionsLaengeHilf*-Funktion ebenfalls rekursiv mit dem neu entstandenen Ausdruck und der Liste unbenutzter Variablennamen aufgerufen, allerdings wird die Schrittzahl unverändert weitergegeben.

```
reduktionsLaenge :: (Eq cname, Eq label) =>
Expr label cname String
-> Integer

reduktionsLaenge expr
= reduktionsLaengeHilf (Just expr) 0 ["_x" ++  show (i :: Integer) | i <- [1..]]
```

Die *reduktionsLaenge*-Funktion selbst besteht nun aus einem einfachen Aufruf der *reduktionsLaengeHilf*-Funktion, mit einem Eingabe-Ausdruck (der die einzige Eingabe für die *reduktionsLaenge*-Funktion ist) und der festen Anfangsschrittzahl 0, sowie einer festen, aber unendlichen Liste von Variablennamen, deren erste Elemente "_x1", "_x2", "_x3", .. lauten, und die als unbenutzte Variablennamen für eine eventuelle Umbenennung dienen sollen (es wird dabei davon ausgegangen, dass die Variablen im Eingabe-Ausdruck keinen dieser Namen haben).

2.7.2 Die *reduktionsLaengeVoll*-Funktion

Die *reduktionsLaengeVoll*-Funktion zählt alle angewendeten Regeln in der Reduktion eines Eingabe-Ausdrucks zu seiner WHNF.

Diese Funktion ist analog der *reduktionsLaenge*-Funktion aufgebaut und verwendet eine Hilfsfunktion, die fast vollständig der *reduktionsLaengeHilf*-Funktion entspricht.

Aus diesem Grund wird an dieser Stelle nur kurz der Unterschied zwischen beiden Hilfsfunktionen erklärt.

```
reduktionsLaengeHilfVoll :: (Eq v, Eq cname, Eq label) =>
Maybe (Expr label cname v) -> Integer -> [v]
-> Integer

reduktionsLaengeHilfVoll expr schritte frischeVars
= case reduziere frischeVars expr of
      (Nothing, frischeVars')
      -> schritte
      (Just (expr', istNormOrdSchritt), frischeVars')
    -> reduktionsLaengeHilfVoll (Just expr') (schritte + 1) frischeVars'
```

Während die *reduktionsLaengeHilf*-Funktion, im Fall, dass der Eingabe-Ausdruck zu einem neuen Ausdruck reduziert werden kann, mit Hilfe des booleschen Wertes der *reduziere*-Funktion kontrolliert, ob eine *lbeta*-Regel angewendet wurde und nur in einem solchen Fall die Schrittzahl um eins erhöht, erhöht die *reduktionsLaengeHilfVoll*-Funktion die Schrittzahl bei jeder erfolgreichen Reduktion, unabhängig davon, welche Regeln dazu verwendet wurden. Dies ergibt also die Gesamtzahl aller angewendeten Regeln.

```
reduktionsLaengeVoll :: (Eq cname, Eq label) =>
Expr label cname String
-> Integer

reduktionsLaengeVoll expr
= reduktionsLaengeHilfVoll (Just expr) 0 ["_x" ++ show (i :: Integer) | i <- [1..]]
```

2.7.3 Die *hatKleinereRLN*-Funktion

Mit Hilfe der soeben definierten *reduktionsLaenge*-Funktion können wir nun auch eine einfache Funktion schreiben, die die `rln` zweier Ausdrücke miteinander vergleicht.

```
hatKleinereRLN:: (Eq cname, Eq label, Eq cname1, Eq label1) =>
Expr label cname String -> Expr label1 cname1 String
-> Bool

hatKleinereRLN expr1 expr2
= if (reduktionsLaenge expr1) <= (reduktionsLaenge expr2)
  then True
  else False
```

Die *hatKleinereRLN*-Funktion erhält als Eingabe zwei LR-Ausdrücke, für die sie jeweils die `rln` berechnet und `True` ausgibt, falls die `rln` des ersten Ausdrucks kleiner oder gleich der `rln` des zweiten Ausdrucks ist, oder `False`, falls dies nicht der Fall ist.

2.7.4 Die *normalForm*-Funktion

Die *normalForm*-Funktion benennt einen LR-Ausdruck konsequent um und berechnet anschließend dessen WHNF.

```
normalFormHilf :: (Eq v, Eq cname, Eq label) =>
Maybe (Expr label cname v) -> [v]
-> Expr label cname v
```

```
normalFormHilf expr frischeVars
= case reduziere frischeVars expr of
        (Nothing, frischeVars')
        -> (fromJust expr)
        (Just (expr', istNormOrdSchritt), frischeVars')
        -> normalFormHilf (Just expr') frischeVars'
```

Auch die *normalFormHilf*-Funktion ruft die *reduziere*-Funktion mit dem Eingabe-Ausdruck und einer Liste unbenutzter Variablennamen auf.

Erhält die *normalFormHilf*-Funktion von der *reduziere*-Funktion ein `Nothing` (wenn die *reduziere*-Funktion den Eingabe-Ausdruck nicht weiter reduzieren kann), so gibt die *normalFormHilf*-Funktion den Eingabe-Ausdruck aus (der ja dann schon in seiner WHNF vorliegt).

Kann die *reduziere*-Funktion den Eingabe-Ausdruck weiter reduzieren, wird die *normalFormHilf*-Funktion rekursiv mit dem neu entstandenen Ausdruck (und der u.U. aktualisierten FriVarListe) aufgerufen.

```
normalForm :: (Eq cname, Eq label) =>
Expr label cname [Char]
-> Expr label cname [Char]

normalForm expr
= normalFormHilf (Just expr') frischeVars
  where (expr', frischeVars)
          = umbenenne expr ["_x" ++ show (I :: Integer) | i <- [1..]]
```

Die *normalForm*-Funktion besteht nun aus einem Aufruf der *normalFormHilf*-Funktion, wobei der Eingabe-Ausdruck zuvor, mit Hilfe der von der *reduktionsLaenge*-Funktion schon bekannten FriVarListe, umbenannt wurde. Die Umbenennung soll hier dazu dienen, mögliche Eingabeungenauigkeiten (wenn Variablennamen im Eingabe-Ausdruck nicht konsequent unterschiedliche Namen haben) zu beseitigen.

Zusammen mit den vorherigen Abschnitten implementieren die oben aufgeführten Funktionen das Programmverhalten, das wir in unseren Zielen festgelegt hatten, nämlich die Berechnung der `rln` (und der `rlnall`) und die WHNF. Außerdem können wir jetzt für zwei Ausdrücke überprüfen, ob der eine Ausdruck eine kleinere `rln` hat als der andere.

Was noch fehlt, ist es, die obigen Funktionen zu testen, um zu sehen, ob sie auch erwartungsgemäß funktionieren.

2.8 Tests

Der folgende Abschnitt soll dazu dienen, die richtige Funktionsweise der im vorherigen Abschnitt vorgestellten Programme (zumindest oberflächlich) zu testen.

Dazu werden im ersten Teil des Abschnitts zwei Beispiel-Ausdrücke zunächst anhand der LR-Reduktionsregeln per Hand reduziert, so dass am Ende jeweils die Anzahl der *lbeta*-Reduktion, die Anzahl aller Regelanwendungen und die WHNF des Ausdrucks explizit angegeben werden können.
Anschließend wird der Ausdruck in die entsprechende Haskell-Repräsentation überführt und die drei Funktionen - *reduktionsLaenge*, *reduktionsLaengeVoll* und *normalForm* - mit dem so repräsentierten Ausdruck aufgerufen und die Ausgabe der Programme mit den erwarteten (per Hand berechneten) Werten verglichen.
Die Beispiel-Ausdrücke werden dabei so gewählt, dass mit möglichst wenigen Beispielen, möglichst viele der implementierten Reduktionsregeln abgedeckt werden.

Im zweiten Teil des Test-Abschnittes sollen die Funktionen dann mit komplexeren Ausdrücken getestet werden. Dazu werden wir arithmetische Ausdrücke mit Hilfe der *Church-Kodierung* in LR-Ausdrücke übersetzen und anschließend für diese wieder die Anzahl der *lbeta*-Regelanwendungen, sowie die Anzahl aller Reduktionsregelanwendungen und die WHNF, mit Hilfe der implementierten Funktionen, bestimmen.

2.8.1 Einfache Testfälle

Der erste Beispiel-Ausdruck lautet

$$((\textbf{letrec}\ x_2 = (x_1\ v),\ x_3 = x_2,\ x_1 = \lambda y.y\ \textbf{in}\ (x_2\ u))\ w)$$

Im folgenden Reduktionsdiagramm stellen die Pfeile jeweils einen Reduktionsschritt dar, wobei die Regel, die zur Reduktion angewendet wurde, links neben dem Pfeil vermerkt ist. Die Anzahl aller Pfeile entspricht demnach der Anzahl aller Regelanwendungen und muss daher der Ausgabe unserer *reduktionsLaengeVoll*-Funktion entsprechen.
Die Zahl neben den Regelnamen zeigt an, ob eine *lbeta*-Reduktion stattgefunden hat, so dass die Anzahl der Einsen der Anzahl der angewendeten *lbeta*-Reduktionen entspricht und mit der Ausgabe unserer *reduktionsLaenge*-Funktion übereinstimmen muss.
Der Ausdruck in der letzten Zeile des Reduktionsdiagramms ist ein Ausdruck, der nicht weiter reduziert werden kann und entspricht damit der WHNF des Beispiel-Ausdrucks und hoffentlich der Ausgabe unserer *normalForm*-Funktion.

$$((\textbf{letrec}\ x_2 = (x_1\ v),\ x_3 = x_2,\ x_1 = \lambda y.y\ \textbf{in}\ (x_2\ u))\ w)$$

$$\downarrow \textit{lapp}\ (+0)$$

$$\textbf{letrec}\ x_2 = (x_1\ v),\ x_3 = x_2,\ x_1 = \lambda y.y\ \textbf{in}\ ((x_2\ u)\ w)$$

$$\downarrow \textit{cp-e}\ (+0)$$

$$\textbf{letrec}\ x_2 = ((\lambda\_x_1.\_x_1)\ v),\ x_3 = x_2,\ x_1 = \lambda y.y\ \textbf{in}\ ((x_2\ u)\ w)$$

$$\downarrow \textit{lbeta}\ (+1)$$

$$\textbf{letrec}\ x_2 = (\textbf{letrec}\ \_x_1 = v\ \textbf{in}\ \_x_1),\ x_3 = x_2,\ x_1 = \lambda y.y\ \textbf{in}\ ((x_2\ u)\ w)$$

$$\downarrow \textit{llet-e}\ (+0)$$

$$\textbf{letrec}\ x_2 = \_x_1,\ x_3 = x_2,\ x_1 = \lambda y.y,\ \_x_1 = v\ \textbf{in}\ ((x_2\ u)\ w)$$

Auf den Ausdruck konnten also, während seiner Reduktion, insgesamt **vier** Reduktionsregeln angewendet werden, wobei **eine** davon eine *lbeta*-Reduktion war.

Die WHNF des ersten Beispiel-Ausdrucks lautet

$$\textbf{letrec}\ x_2 = \_x_1,\ x_3 = x_2,\ x_1 = \lambda y.y,\ \_x_1 = v\ \textbf{in}\ ((x_2\ u)\ w)$$

Um den Ausdruck nun in unsere Funktionen eingeben zu können, müssen wir ihn zunächst in unsere Datentyp-Repräsentation übersetzen.

Dies ergibt folgenden Ausdruck

```
beispiel1 = App (Letrec ["x2" :=: App (Var "x1") (Var "v"),
                         "x3" :=: Var "x2",
                         "x1" :=: Lam "y" (Var "y")]
                        (App (Var "x2") (Var "u")))
                (Var "w")
```

Geben wir diesen nun in unsere Funktionen ein, erhalten wir folgende Werte

```
*Language> reduktionsLaengeVoll beispiel1
4
*Language> reduktionsLaenge beispiel1
1
*Language> normalForm beispiel1
Letrec ["_x5" :=: Var "v",
        "_x1" :=: Var "_x5",
        "_x2" :=: Var "_x1",
        "_x3" :=: Lam "_x4" (Var "_x4")]
       (App (App (Var "_x1") (Var "u")) (Var "w"))
```

Wir sehen, dass alle Werte den per Hand berechneten Werten entsprechen (bis auf Umbenennung der Normalform) und wir daher davon ausgehen können, dass die von uns implementierten Reduktionsregeln (*lapp*, *cp-e*, *lbeta* und *llet-e*) korrekt funktioniert haben.
Von den insgesamt sechs zu implementierenden Reduktionsregeln wurden zwei (*llet-in* und *cp-in*) allerdings noch nicht angewendet und somit auch noch nicht überprüft.

Dies wollen wir jetzt mit unserem zweiten Beispiel-Ausdruck machen.

Der zweite Beispiel-Ausdruck lautet

$$\textbf{letrec}\ x_1 = \lambda x.x,\ x_2 = x_1\ \textbf{in}\ (\textbf{letrec}\ x_3 = x_2\ \textbf{in}\ (x_3\ y))$$

Das Reduktionsdiagramm für den zweiten Beispiel-Ausdruck ist:

$$\textbf{letrec}\ x_1 = \lambda x.x,\ x_2 = x_1\ \textbf{in}\ (\textbf{letrec}\ x_3 = x_2\ \textbf{in}\ (x_3\ y))$$

$$\downarrow \textit{llet-in}\ (+0)$$

$$\textbf{letrec}\ x_1 = \lambda x.x,\ x_2 = x_1,\ x_3 = x_2\ \textbf{in}\ (x_3\ y)$$

$$\downarrow \textit{cp-in}\ (+0)$$

$$\textbf{letrec}\ x_1 = \lambda x.x,\ x_2 = x_1,\ x_3 = x_2\ \textbf{in}\ ((\lambda\_x_1.\_x_1)\ y)$$

$$\downarrow \textit{lbeta}\ (+1)$$

$$\textbf{letrec}\ x_1 = \lambda x.x,\ x_2 = x_1,\ x_3 = x_2\ \textbf{in}\ (\textbf{letrec}\ \_x_1 = y\ \textbf{in}\ \_x_1)$$

$$\downarrow \textit{llet-in}\ (+0)$$

$$\textbf{letrec}\ x_1 = \lambda x.x,\ x_2 = x_1,\ x_3 = x_2,\ \_x_1 = y\ \textbf{in}\ \_x_1$$

Auf den zweiten Beispiel-Ausdruck konnten also wieder insgesamt **vier** Reduktionsregeln angewendet werden, von denen wieder **eine** eine *lbeta*-Reduktion war.

Die WHNF des zweiten Beispiel-Ausdrucks lautet

$$\textbf{letrec}\ x_1 = \lambda x.x,\ x_2 = x_1,\ x_3 = x_2,\ \_x_1 = y\ \textbf{in}\ \_x_1$$

Die Repräsentation des zweiten Beispiel-Ausdrucks in unserer Syntax ergibt

```
beispiel2 = Letrec ["x1" :=: Lam "x" (Var "x"),
                    "x2" :=: Var "x1"]
                   (Letrec ["x3" :=: Var "x2"]
                           (App (Var "x3") (Var "y")))
```

Die Ergebnisse nach Eingabe dieses Ausdruckes in unsere Funktionen ergibt

```
*Language> reduktionsLaengeVoll beispiel2
4
*Language> reduktionsLaenge beispiel2
1
*Language> normalForm beispiel2
Letrec ["_x1" :=: Lam "_x3" (Var "_x3"),
        "_x2" :=: Var "_x1",
        "_x4" :=: Var "_x2",
        "_x5" :=: Var "y"]
       (Var "_x5")
```

Wieder können wir sehen, dass alle per Computer berechneten Werte den per Hand berechneten Werten entsprechen und daher können wir auch für die restlichen Reduktionsregeln davon ausgehen, dass sie korrekt funktionieren.

Da wir bei der Reduktion dieser beiden Beispiele alle Regeln mindestens ein Mal angewendet haben, können wir annehmen, dass unsere Funktionen korrekt funktionieren.

2.8.2 Komplexe Testfälle

Im zweiten Teil des Tests-Abschnittes wollen wir die so getesteten Funktionen auf komplexere LR-Ausdrücke anwenden.

Dazu bedienen wir uns der Church-Kodierungen und definieren uns damit zunächst verschiedene Programm-Konstrukte (und Operationen), sowie Objekte auf denen wir operieren können, nämlich die ganzen Zahlen.

Auf diese Weise erhalten wir relativ einfach komplexe LR-Ausdrücke, die einfache Programmaufrufe repräsentieren.
Mit unseren Funktionen können wir dann die Reduktionslängen dieser Programme berechnen und kommen damit dem eigentlich angepeilten Anwendungsbereich - dem Vergleich der Laufzeit von Haskell-Programmen, ausgedrückt in dessen Kernsprache - ein kleines bißchen näher.

Zunächst definieren wir uns die natürlichen Zahlen (inklusive der Null) folgendermaßen:

```
zero = Lam "f" (Lam "x" (Var "x"))

number 0 = zero
number n = App successor (number (n-1))

successor
= Lam "n" (Lam "f" (Lam "x" (App (Var "f") (App (App (Var "n")
                                                      (Var "f"))
                                                 (Var "x")))))
```

Dann definieren wir uns die Wahrheitswerte und das If-Then-Else:

```
true          = Lam "a" (Lam "b" (Var "a"))
false         = Lam "a" (Lam "b" (Var "b"))
ifthenelse b t e = App (App b t) e
```

Anschließend definieren wir uns einfache Operationen auf den Zahlen:

```
minus = Lam "m" $ Lam "n" $ App (App (Var "n") prede )
                                (Var "m")

prede
= Lam "n" $ Lam "f" $ Lam "x" $
  (App (App (App (Var "n")
                 (Lam "g" $ Lam "h" $ App (Var "h")
                                          (App (Var "g")
                                               (Var "f"))))
             (Lam "u" $ Var "x"))
        (Lam "u" $ Var "u"))

add x y = App (App plus x) y

plus
= Lam "m" (Lam "n" (Lam "f" (Lam "x" (App (App (Var "m")
                                               (Var "f"))
                                          (App (App (Var "n")
                                                    (Var "f"))
                                               (Var "x"))))))

isZero = Lam "n" (App (App (Var "n") (Lam "x" false))
                      (true))
```

Abschließend bauen wir uns aus dem zuvor Definiertem eine einfache Funktion zusammen, die eine Zahl als Eingabe erhält und von ihrer Eingabe so lange Eins abzieht, bis die Eingabe Null ist:

```
evalNum x
= Letrec
  ["evalNum" :=: Lam "n"
                    (ifthenelse
                        (App isZero (Var "n"))
                         true
                        (App (Var "evalNum") (App prede (Var "n")))) ]
 (App (Var "evalNum") (x))
```

All diese Konstrukte kombinieren wir nun und verwenden sie als Eingaben für unsere Funktionen.

```
*Language> reduktionsLaengeVoll (add (number 3) (number 4))
4
*Language> reduktionsLaenge (add (number 3) (number 4))
2
*Language> normalForm (add (number 3) (number 4))
Letrec ["_x1" :=: App (Lam "_x5" (Lam "_x6" (Lam "_x7" (App (Var "_x6") (App (App (Var
"_x5") (Var "_x6")) (Var "_x7")))))) (App (Lam "_x8" (Lam "_x9" (Lam "_x10" (App (Var
"_x9") (App (App (Var "_x8") (Var "_x9")) (Var "_x10")))))) (App (Lam "_x11" (Lam
"_x12" (Lam "_x13" (App (Var "_x12") (App (App (Var "_x11") (Var "_x12")) (Var
"_x13")))))) (Lam "_x14" (Lam "_x15" (Var "_x15"))))),"_x2" :=: App (Lam "_x16" (Lam
"_x17" (Lam "_x18" (App (Var "_x17") (App (App (Var "_x16") (Var "_x17")) (Var
"_x18")))))) (App (Lam "_x19" (Lam "_x20" (Lam "_x21" (App (Var "_x20") (App (App (Var
"_x19") (Var "_x20")) (Var "_x21")))))) (App (Lam "_x22" (Lam "_x23" (Lam "_x24" (App
(Var "_x23") (App (App (Var "_x22") (Var "_x23")) (Var "_x24")))))) (App (Lam "_x25"
(Lam "_x26" (Lam "_x27" (App (Var "_x26") (App (App (Var "_x25") (Var "_x26")) (Var
"_x27")))))) (Lam "_x28" (Lam "_x29" (Var "_x29")))))] (Lam "_x3" (Lam "_x4" (App
(App (Var "_x1") (Var "_x3")) (App (App (Var "_x2") (Var "_x3")) (Var "_x4")))))
```

```
*Language> reduktionsLaengeVoll (evalNum (App (App minus
(number 5)) (number 3)))
116
*Language> reduktionsLaenge (evalNum (App (App minus (number 5))
(number 3)))
34

*Language> reduktionsLaengeVoll (evalNum (add (add (number 23)
(number 12)) (App (App minus (number 13)) (number 9))))
171
*Language> reduktionsLaenge (evalNum(add (add (number 23)
(number 12)) (App (App minus (number 13)) (number 9))))
54

*Language> reduktionsLaengeVoll (evalNum (add (number 1342)
(number 235)))
2806
*Language> reduktionsLaenge (evalNum (add (number 1342)
(number 235)))
1372
```

Dieser Auszug stellt den Abschluss des Tests-Abschnitts dar.

Die Tests sollten uns überzeugen, dass unsere Funktionen korrekt definiert wurden und das richtige Programmverhalten zeigen. Die komplexen Testfälle sollten uns außerdem zeigen, wie der Weg von der Reduktion einfacher Ausdrücke bis zum Vergleich ganzer Programme (in Form von LR-Ausdrücken) aussehen könnte.

Im folgenden Ausblick soll kurz angedeutet werden, wie dieser Weg weitergegangen werden könnte.

3 Ausblick

In diesem abschließenden Abschnitt sollen kurz einige Möglichkeiten erwähnt werden, das vorliegende Programm zu erweitern, verbessern und anzuwenden.

Ein offensichtlicher, erster Schritt wäre es, das hier vorgestellte Programm auf die vollständige Syntax und damit auch um die fehlenden Reduktionsregeln zu erweitern. Das hieße konkret, die Syntax um Seq-, Case- und Cons-Ausdrücke zu erweitern und die Regeln *lcase*, *lseq*, *seq-c*, *seq-in*, *seq-e*, *case-c*, *case-in* und *case-e* zu implementieren.

Das vorliegende Programm testet bisher nur, ob für zwei LR-Ausdrücke s und t die Ungleichung `rln`(s) ≤ `rln`(t) gilt. Um testen zu können, ob eine Programmtransformation ein Improvement ist, müsste man für alle Kontexte C testen, ob `rln`(C[s]) ≤ `rln`(C[t]) gilt. Dies ist nicht so ohne Weiters möglich. Man könnte allerdings mit einer effizienten Auflistung aller Kontexte und anschließendem Einsetzen der beiden Eingabe-Ausdrücke versuchen, Gegenbeispiele dafür zu finden, dass der eine Eingabe-Ausdruck ein Improvement des anderen ist (also dass ein Kontext existiert, so dass `rln`(C[s]) > `rln`(C[t])).

Eine weitere Erweiterung könnte darin bestehen, Sharing von ausgewerteten Unterausdrücken zu implementieren. Wenn während einer Normalordnungsreduktion Unterausdrücke vervielfacht werden, kann Sharing die Reduktion wesentlich verkürzen.

Eine andere Möglichkeit, das Programm zu erweitern, wäre es, die Funktionen für getypte Lambda-Kalkuli umzuschreiben. Dies wäre notwendig dafür, bestimmte Programmtransformation als Improvements nachweisen zu können.

Jede dieser Erweiterungen bietet die Möglichkeit, das Programm kompakter zu gestalten oder zumindest einzelne Funktionen allgemeiner zu implementieren, um die Übersichtlichkeit und Wartbarkeit des Programms zu verbessern.

Außerdem kommt es regelmäßig vor, dass ein Programm Denk- und Programmier-Fehler enthält, die nur durch umfassendes Testen des Programms mit zahlreichen, unterschiedlichen Beispielen (idealerweise aus unterschiedlichen Anwendungsbereichen) entdeckt und anschließend behoben werden können.

Hätte man erst einmal ein Programm, mit dem man auf Improvements (oder Gegenbeispiele zu vermeintlichen Improvements) testen kann, fänden sich in der Industrie und der Software-Technologie sicherlich zahlreiche Anwendungs-möglichkeiten – von einfachen Programmierwerkzeugen bis voll automatisierter Programmgenerierung.

Aber auch die Wissenschaft hätte sicher ihren Spaß an Programmen, die immer bessere Programme schreiben.

4 Literatur

1. ABRAMSKY, S.,
The Lazy Lambda Calculus,
Research topics in functional programming,
Addison-Wesley, 1990, pp 65 - 116,

2. ALAMA, J.,
The Lambda Calculus,
The Stanford Encyclopedia of Philosophy (Spring 2015 Edition),
Edward N. Zalta (ed.),
<http://plato.stanford.edu/archives/spr2015/entries/lambda-calculus/>.

3. ALLEN, C., MORONUKI, J.,
Haskell Programming from first principles,
(Version Juli 2015), book-in-progress (www.haskellbook.com)

4. ARIOLA, Z. M., FELLEISEN, M., MARAIST, J., ODERSKY, M., WADLER, P.,
A Call-By-Need Lambda Calculus,
POPL '95 Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp 233-246

5. BARENDREGT, H. P.,
Lambda Calculi with Types,
Handbook of Logic in Computer Science, Volume II, 1992

6. BARENDREGT, H. P.,
The impact of the lambda calculus in logic and computer science,
The Bulletin of Symbolic Logic, Vol. 3, No. 2 (Jun., 1997), pp. 181-215

7. BARENDREGT, H. P., BARENDSEN, E.,
Introduction to Lambda Calculus,
www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf, 2000

8. BARENDREGT, H. P.,
Lambda Calculus, its Syntax and Semantics,
Studies in Logic, Volume 40, College Publications, 2012

9. BARKER-PLUMMER, D.,
Turing Machines,
The Stanford Encyclopedia of Philosophy (Summer 2013 Edition),
Edward N. Zalta (ed.),
<http://plato.stanford.edu/archives/sum2013/entries/turing-machine/>.

10. BIRD, R.,
Thinking Functionally with Haskell,
Cambridge University Press, 2015

11. CHANG, S., FELLEISEN, M.,
The Call-by-need Lambda Calculus, Revisited,
Programming Languages and Systems,
Volume 7211 of the series Lecture Notes in Computer Science, 2012, pp 128-147

12. FELLEISEN, M., FLATT, M.,
Programming Languages and Lambda Calculi,
Vorlesungsunterlagen, 2006,
www.cs.utah.edu/~mflatt/past-courses/cs7520/public_html/s06/notes.pdf

13. HUDAK, P.,
The Haskell School of Expression,
Cambridge University Press, 2007

14. HUGHES, J.,
Why Functional Programming Matters,
in "Research Topics in Functional Programming", ed. D. Turner,
Addison-Wesley, 1990, pp 17-42

15. HUTTON, G.,
Programming in Haskell,
Cambridge University Press, 2007

16. LIPOVAČA, M.,
Learn You A Haskell For Great Good! A Beginner's Guide,
No Starch Press, San Francisco, 2011

17. MEIJER, E., DRAYTON, P.,
Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages,
citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.5966&rep=rep1&type=pdf

18. MICHAELSON, G.,
An Introduction to Functional Programming through Lambda Calculus,
Addison-Wesley, 1988
www.macs.hw.ac.uk/~greg/books/

19. O'SULLIVAN, B., GOERZEN, J., STEWART, D.,
Real World Haskell,
O'Reilly, 2009

20. PEYTON JONES, S. L.,
The Implementation of Functional Programming Languages,
Prentice Hall International, 1987

21. PLOTKIN, G. D.,
Call-By-Name, Call-By-Value and the Lambda-Calculus,
Theoretical Computer Science, 1 (1975), 125-159

22. ROJAS, R.,
A Tutorial Introduction to the Lambda Calculus,
Vorlesungsunterlagen (Wintersemester 1997/98),
www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

23. SABEL, D.,
Anleitung zum ***Praktikum Funktionale Programmierung***,
(Version Sommersemester 2015),
www.ki.informatik.uni-frankfurt.de/lehre/SS2015/FP-PR/anleitung.pdf

24. SCHMIDT-SCHAUSS, M., SABEL, D., SCHÜTZ, M.,
Safety of Nöcker's strictness analysis,
JFP (Journal of Functional Programming), 18 (4): 503–551, 2008.

25. SCHMIDT-SCHAUSS, M., SABEL, D.,
Skript ***Grundlagen der Programmierung 2***,
Kapitel 1 ***Programmiersprachen: rekursives Programmieren in Haskell***,
(Version Sommersemester 2014),
www-stud.informatik.uni-frankfurt.de/~prg2/SS2014/skript/teil1/Kap-1u2.pdf

26. SCHMIDT-SCHAUSS, M., SABEL, D.,
Skript ***Einführung in die Funktionale Programmierung***,
(Version Wintersemester 2014/2015)
www.ki.informatik.uni-frankfurt.de/lehre/WS2014/EFP/skript/skript.pdf

27. SCHMIDT-SCHAUSS, M., SABEL, D.,
Improvements in a Functional Core Language with Call-By-Need Operational Semantics,
Technical Report Frank-55 (28. März 2015)
www.ki.informatik.uni-frankfurt.de/papers/frank/frank-55.pdf

28. SCHMIDT-SCHAUSS, M., SABEL, D.,
Sharing-Aware Improvements in a Call-by-Need Functional Core Language,
(2015, Unveröffentlicht)

29. SCHMIDT-SCHAUSS, M., SABEL, D.,
Sharing Decorations for Improvements in a Functional Core Language with Call-By-Need Operational Semantics,
Technical Report Frank-56, 2015
www.ki.informatik.uni-frankfurt.de/papers/frank/frank-56.pdf

30. SELINGER, P.,
Lecture Notes on the Lambda Calculus,
Vorlesungsunterlagen, 2013,
www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf

31. THOMPSON, S.,
Haskell. The craft of functional programming,
Addison-Wesley, 2011

32. WADLER, P.,
The essence of functional programming,
Invited talk at 19th Annual Symposium on Principles of Programming Languages, 1992, www.eliza.ch/doc/wadler92essence_of_FP.pdf

33. WAGENKNECHT, C.,
Programmierparadigmen. Eine Einführung auf der Grundlage von Scheme,
Teubner Verlag, 2004

5 Anhang

Hier findet sich der vollständige Quellcode aller beschriebenen Funktionen (inklusive dem Quellcode der lediglich umschriebenen Hilfsfunktionen).

```
--Die Datentyp-Deklaration-------------------------------------------------

module Language where
import Data.List
import Data.Char
import Data.Maybe

data Expr label cname v =
   Var v
 | Lam v (Expr label cname v)
 | App (Expr label cname v) (Expr label cname v)
 | Letrec [Binding label cname v] (Expr label cname v)
 | Seq (Expr label cname v) (Expr label cname v)
 | Cons cname [Expr label cname v]
 | Case (Expr label cname v) [CAlt label cname v] -}
 | Label label (Expr label cname v)
 deriving(Eq,Show)

data CAlt label cname v =
 CAlt cname [v] (Expr label cname v)
 deriving(Eq,Show) -}

data Binding label cname v =
 v :=: (Expr label cname v)
 deriving(Eq,Show)
```

```
--Die reduziere-Funktionen-------------------------------------------------

reduziere :: (Eq v, Eq cname, Eq label) =>
             [v] -> Maybe (Expr label cname v)
             -> (Maybe (Expr label cname v, Bool), [v])

reduziere frischeVars (Just (Var x))
= (Nothing, frischeVars)

reduziere frischeVars (Just (Lam x e))
= (Nothing, frischeVars)

reduziere frischeVars (Just (App (Letrec b e) f))
= (Just (Letrec b (App e f), False), frischeVars)

reduziere frischeVars (Just (App (Lam x e) f))
= (Just (Letrec [x :=: f] e, True), frischeVars)

reduziere frischeVars (Just (App e f))
= let (e1, frischeVars') = (reduziere frischeVars (Just e))
  in case e1 of
          Nothing
          -> (Nothing, frischeVars')
          Just (e2, istNormOrdSchritt)
          -> (Just (App e2 f, istNormOrdSchritt), frischeVars')

reduziere frischeVars (Just (Letrec b (Letrec c e)))
= (Just (Letrec (b ++ c) e, False), frischeVars)

reduziere frischeVars (Just (Letrec b (Lam x e)))
= (Nothing, frischeVars)

reduziere frischeVars (Just (Letrec b e))
= case (reduziere frischeVars (Just e)) of
       (Just (e1, istNormOrdSchritt), frischeVars')
        -> (Just (Letrec b e1, istNormOrdSchritt), frischeVars')
       (Nothing, frischeVars')
        -> case (findeVar (Just e)) of
                Just (x, context)
                -> hilf frischeVars' b e x (\z -> Letrec b (context z))
                Nothing
                -> (Nothing, frischeVars')

reduziere frischeVars Nothing
= (Nothing, frischeVars)
```

```
--Die findeVar-Funktion-----------------------------------------------------

findeVar :: Maybe (Expr label cname v)
            -> Maybe (v, Expr label cname v -> Expr label cname v)

findeVar (Just (Lam x e))
= Nothing

findeVar (Just (Letrec b e))
= Nothing

findeVar (Just (Var y))
= Just (y, id)

findeVar (Just (App e f))
= case (findeVar (Just e)) of
        Nothing
        -> Nothing
        Just (y, context)
        -> Just (y, \z -> App (context z) f)

findeVar Nothing
= Nothing
```

```
--Die hilf-Funktion-------------------------------------------------------

hilf :: (Eq v, Eq cname, Eq label) =>
        [v] -> [Binding label cname v] ->
        Expr label cname v -> v -> (Expr label cname v -> Expr label cname v)
        -> (Maybe (Expr label cname v,Bool),[v])

hilf frischeVars b e x context
= let r = (exprVonBind x b)
  in case r of
          Just (Letrec b1 e1)
          -> let b2 = (aktualisiereBind e1 x b)
             in (Just (Letrec (b1 ++ b2) e, False), frischeVars)
          Just (Var z)
          -> hilf frischeVars b e z context
          Just (Lam a e')
          -> let (umbenannteAbstr, frischeVars')= umbenenne (Lam a e') frischeVars
             in (Just ((context umbenannteAbstr), False), frischeVars')
          otherwise
          -> case (reduziere frischeVars r) of
                  (Just (r1, istNormOrdSchritt), frischeVars')
                   -> let b1 = (aktualisiereBind r1 x b)
                      in (Just (Letrec b1 e, istNormOrdSchritt), frischeVars')
                  Nothing, frischeVars')
                  -> case (findeVar r) of
                          Nothing
                          -> (Nothing, frischeVars')
                          Just (y, context1)
                          -> hilf frischeVars' b e y (\z -> Letrec ((x :=: (context1 z)):
                                                              (löscheBind x b)) e)
```

```
--Die umbenenneHilf-Funktion-----------------------------------------------

umbenenneHilf :: (Eq v) =>
                 [(v, v)] -> Expr label cname v -> [v]
                 -> (Expr label cname v, [v])

umbenenneHilf belegungs (Var x) frischeVars
= case lookup x belegungs of
       Nothing
       -> (Var x, frischeVars)
       Just y
       -> (Var y, frischeVars)

umbenenneHilf belegungs (Lam x e) (f : frischeVars)
= let (e', frischeVars') = umbenenneHilf ((x, f) : belegungs) e frischeVars
  in (Lam f e', frischeVars')

umbenenneHilf belegungs (App e1 e2) frischeVars
= let (e1', frischeVars')  = umbenenneHilf belegungs e1 frischeVars
      (e2', frischeVars'') = umbenenneHilf belegungs e2 frischeVars'
  in (App e1' e2', frischeVars'')

umbenenneHilf belegungs (Letrec b e) frischeVars
= let bindvars             = map (\(x :=: r) -> x) b
      l                    = length bindvars
      umbenennung          = zip bindvars frischeVars
      frischeVars'         = drop l frischeVars
      belegungs'           = umbenennung ++ belegungs
      (b', frischeVars'')  = umbenenneBinds belegungs' b frischeVars'
      (e', frischeVars''') = umbenenneHilf belegungs' e frischeVars''
  in (Letrec b' e', frischeVars''')
```

```
--Die umbenenneBinds-Funktion-----------------------------------------------

umbenenneBinds :: (Eq v) =>
                  [(v, v)] -> [Binding label cname v] -> [v]
                  -> ([Binding label cname v], [v])

umbenenneBinds _ [] frischeVars
= ([], frischeVars)

umbenenneBinds belegungs ((x :=: e) : xs) frischeVars
= let (e', frischeVars')   = umbenenneHilf belegungs e frischeVars
      (xs', frischeVars'') = umbenenneBinds belegungs xs frischeVars'
      x'                   = fromJust (lookup x belegungs)
  in ((x' :=: e') : xs', frischeVars'')



--Die umbenenne-Funktion-----------------------------------------------------

umbenenne :: (Eq v) =>
             Expr label cname v -> [v] ->
            (Expr label cname v, [v])

umbenenne expr vars
= umbenenneHilf [] expr vars
```

```
--Die exprVonBind-Funktion-------------------------------------------------

exprVonBind :: (Eq v, Eq cname, Eq label) =>
               v -> [Binding label cname v]
               -> Maybe (Expr label cname v)

exprVonBind x []
= Nothing

exprVonBind x ((y :=: r) : ys)
= if (x == y) then (Just r)
  else exprVonBind x ys



--Die aktualisiereBind-Funktion---------------------------------------------

aktualisiereBind :: (Eq v, Eq cname, Eq label) =>
                    Expr label cname v -> v -> [Binding label cname v]
                    -> [Binding label cname v]

aktualisiereBind neu var []
= []

aktualisiereBind neu var ((x :=: expr) : xs)
= if (x == var) then (x :=: neu) : xs
  else (x :=: expr) : aktualisiereBind neu var xs



--Die löscheBind-Funktion---------------------------------------------------

löscheBind :: (Eq v, Eq cname, Eq label) =>
              v -> [Binding label cname v]
              -> [Binding label cname v]

löscheBind x []
= []

löscheBind x ((y :=: e2) : ys)
= if (x == y) then ys
  else (y :=: e2) : (löscheBind x  ys)
```

```
--Die reduktionsLaengeHilf-Funktion----------------------------------------

reduktionsLaengeHilf :: (Eq v, Eq cname, Eq label) =>
                        Maybe (Expr label cname v) -> Integer -> [v]
                        -> Integer

reduktionsLaengeHilf expr schritte frischeVars
= case reduziere frischeVars expr of
       (Nothing, frischeVars')
       -> schritte
       (Just (expr', istNormOrdSchritt), frischeVars')
       -> if istNormOrdSchritt
          then reduktionsLaengeHilf (Just expr') (schritte + 1) frischeVars'
          else reduktionsLaengeHilf (Just expr') schritte frischeVars'



--Die reduktionsLaenge-Funktion---------------------------------------------

reduktionsLaenge :: (Eq cname, Eq label) =>
                    Expr label cname String
                    -> Integer

reduktionsLaenge expr
= reduktionsLaengeHilf (Just expr) 0 ["_x" ++ show (i :: Integer) | i <- [1..]]



--Die hatKleinereRLN-Funktion-----------------------------------------------

hatKleinereRLN :: (Eq cname, Eq label, Eq cname1, Eq label1) =>
                  Expr label cname String -> Expr label1 cname1 String
                  -> Bool

hatKleinereRLN expr1 expr2
= if (reduktionsLaenge expr1) < (reduktionsLaenge expr2) then True
  else False
```

```
--Die reduktionsLaengeHilfVoll-Funktion-------------------------------------

reduktionsLaengeHilfVoll :: (Eq v, Eq cname, Eq label) =>
                            Maybe (Expr label cname v) -> Integer -> [v]
                            -> Integer

reduktionsLaengeHilfVoll expr schritte frischeVars
= case reduziere frischeVars expr of
       (Nothing, _)
       -> schritte
       (Just (expr', istNormOrdSchritt), frischeVars')
       -> reduktionsLaengeHilfVoll (Just expr') (schritte + 1) frischeVars'



--Die reduktionsLaengeVoll-Funktion-----------------------------------------

reduktionsLaengeVoll :: (Eq cname, Eq label) =>
                        Expr label cname String
                        -> Integer

reduktionsLaengeVoll expr
= reduktionsLaengeHilfVoll (Just expr) 0 ["_x" ++ show (i :: Integer) | i <- [1..]]
```

```
--Die normalFormHilf-Funktion-----------------------------------------------

normalFormHilf :: (Eq v, Eq cname, Eq label) =>
                  Maybe (Expr label cname v) -> [v]
                  -> Expr label cname v

normalFormHilf expr frischeVars
= case reduziere frischeVars expr of
       (Nothing, frischeVars')
       -> (fromJust expr)
       (Just (expr', istNormOrdSchritt), frischeVars')
       -> normalFormHilf (Just expr') frischeVars'



--Die normalForm-Funktion-----------------------------------------------------

normalForm :: (Eq cname, Eq label) =>
              Expr label cname [Char] ->
              Expr label cname [Char]

normalForm expr
= normalFormHilf (Just expr') frischeVars
  where (expr', frischeVars) = umbenenne expr ["_x" ++ show (i :: Integer) | i <- [1..]]
```

6 Erklärung

Ich erkläre hiermit,
dass ich die Diplomarbeit selbstständig verfasst
und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Frankfurt, den 29.09.2015