Fachbereich 12 Informatik und Mathematik
Institut für Informatik

Masterarbeit

Konzeption und Implementierung eines Werkzeugs zur automatischen Überprüfung von Programmoptimierungen in funktionalen Programmiersprachen

Tommaso Castrovillari

15. April 2016

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Professur für Künstliche Intelligenz und Softwaretechnologie

Selbstständigkeitserklärung gemäß §24 Abschnitt 12 der Masterordnung

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

Frankfurt, den 15. April 2016

Tommaso Castrovillari

Zusammenfassung

Mit Programmtransformationen lässt sich Programmcode in semantisch äquivalenten Code transformieren. Wenn das dadurch resultierende Programm weniger Laufzeit oder Speicherplatz benötigt, dann handelt es sich bei der Transformation um eine Programmoptimierung. In dieser Arbeit wird ein Werkzeug entwickelt, um Programmoptimierungen in funktionalen Programmiersprachen zu überprüfen. Die Überprüfung konzentriert sich hierbei auf die Anzahl der Reduktionsschritte bei der Auswertung eines funktionalen Ausdrucks.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seit der Verwandlung der Computer von riesigen Maschinen in den Kellern weniger großer Firmen hin zum praktischen Wegbegleiter in sämtlichen Formen, sind diese kleinen und großen Helfer aus unserem Alltag nicht mehr wegzudenken. Das dadurch resultierende verstärkte Interesse nach funktionierender und effizienter Software hat zu einer Vielzahl an Entwicklungen im Bereich der Programmiersprachen geführt. Die Anzahl an dokumentierten Programmiersprachen liegt mittlerweile bei über 2500 [OM16] und ein großer Teilbereich der Informatik beschäftigt sich mit diesen.

Solche Entwicklungen sind nur möglich, weil aktiv an der Verbesserung der Programmiersprachen und an der Codeerzeugung geforscht wird. Einer der wichtigsten Prozesse bei der Codeerzeugung ist die Übersetzung und Optimierung des Programmcodes. Dabei wird der Programmcode analysiert und es wird versucht, die Laufzeit und den Speicherplatzbedarf zu reduzieren. Der resultierende optimierte Programmcode und das ursprüngliche Programm müssen sich hierbei exakt gleich verhalten. Das bedeutet die Semantik des Programms darf bei diesem Prozess nicht verändert werden. Automatisierte Programmoptimierungen sind notwendig, um den Aufwand bei der manuellen Optimierung von Code zu reduzieren. Zudem führen diese Optimierungen durch die Reduzierung der Größe des Programms, die Laufzeitoptimierung und die Reduzierung des Speicherplatzbedarfs dazu, dass Software mit immer weniger Ressourcen auskommt.

Mit dem Lambda-Kalkül legte Alonzo Church 1936 das Fundament für die ersten Programmiersprachen und für die funktionale Programmierung. Heutzutage verwenden immer mehr bekannte Firmen wie Facebook funktionale Programmiersprachen, um ihre Anforderungen umzusetzen. [Mar15] Mit funktionalen Programmiersprachen lässt sich mit relativ einfachen Mitteln paralleler, modularer und wartungsfreundlicher Code schreiben. Deshalb setzten sich immer mehr Konzepte aus der funktionalen

Programmierung auch bei nicht-funktionalen Sprachen durch, wie die Einführung von Lambda-Ausdrücken in Java 8 zeigt.

Das Thema dieser Arbeit ist die Entwicklung eines Werkzeugs, um Programmoptimierungen in funktionalen Programmiersprachen zu untersuchen. Hierfür wird ein erweiterter Lambda-Kalkül vorgestellt und implementiert. Mit diesem lassen sich funktionale Programme schreiben und auswerten. Die Überprüfung von Programmoptimierungen konzentriert sich hierbei auf die Anzahl an Reduktionsschritte die nötig sind, um ein solches Programm auszuwerten.

Hierzu werden in Kapitel 2 zunächst die Grundlagen für das Thema dieser Arbeit eingeführt. Es wird ein erweiterter Lambda-Kalkül vorgestellt zusammen mit den Grundlagen der Auswertung eines funktionalen Programms.
In Kapitel 3 werden die Anforderungen an das Werkzeug definiert. Zudem wird das Werkzeug konzipiert und es wird der Rahmen für die Implementierung gesetzt.
In Kapitel 4 wird das erarbeitete Konzept umgesetzt und es wird die Implementierung erklärt.
In Kapitel 5 wird das fertige Werkzeug anhand von kleinen Anwendungsbeispielen vorgestellt. Es wird eine zusätzliche Programmtransformation implementiert und es wird eine Analyse von zwei Ausdrücken durchgeführt.
Die Arbeit wird dann in Kapitel 6 mit einem Fazit abgeschlossen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen für das Thema dieser Arbeit vermittelt. Hierzu gehört eine Einführung in die funktionale Programmierung in Abschnitt 2.1, eine Vertiefung in die Auswertung in Abschnitt 2.2 und in die Analyse von funktionalen Programmen in Abschnitt 2.3. Dieses Kapitel gibt dabei grundlegende Erklärungen aus [SSS14] und [SS15] wieder.

2.1 Funktionale Programmierung

Programmiersprachen lassen sich grundsätzlich in *deklarative* und *imperative* Programmiersprachen unterteilen. Diese Programmierparadigmen unterscheiden sich in wesentlichen Punkten voneinander.
Imperative Programmiersprachen lassen sich als eine Folge von Anweisungen lesen, die sequentiell abgearbeitet werden müssen. Genauso wie ein Kochrezept oder eine Aufbauanleitung für ein Möbelstück beschreiben imperative Programmiersprachen wie ein Resultat erzeugt werden soll. Dabei ist es durchaus üblich, dass imperative Programmiersprachen den Speicher des Rechners und den internen Zustand verändern.
Deklarative Programmiersprachen hingegen beschreiben *was* berechnet werden soll. Es gibt eine Vielzahl an deklarativen Programmiersprachen, die in verschiedene Kategorien eingeteilt werden können. Zwei davon sind die logischen Programmiersprachen und die funktionalen Programmiersprachen.
Programme die mit einer logischen Programmiersprache geschrieben wurden sind eine Sammlung von logischen Formeln und Fakten, die mit Hilfe von logischen Schlüssen neue Fakten herleiten können.
Funktionale Programme hingegen sind eine Sammlung von mathematischen Funktionsdefinitionen, die sich am Ende zu einem einzigen Wert auswerten lassen. Im Idealfall passiert diese Auswertung ohne Veränderung des Speichers, dass heißt ohne *Seiteneffekte*.

Ein weiteres wichtiges Prinzip der funktionalen Programmiersprachen ist die Eigenschaft der *referentiellen Transparenz* [SSS14, S. 2]:

Die Anwendung einer gleichen Funktion auf gleiche Argumente liefert stets das gleiche Resultat. Variablen in funktionalen Programmiersprachen bezeichnen keine Speicherplätze, sondern stehen für (unveränderliche) Werte.

Programmiersprachen die diese Eigenschaft erfüllen werden im Allgemeinen als *rein* bezeichnet. Diese Eigenschaften ermöglichen es Probleme aus einem anderen Blickwinkel zu betrachten und sich bei der Lösung dieser auf die zu erwartende Antwort zu konzentrieren. Die Verwendung von Funktionen erlaubt es Probleme zu abstrahieren und durch entsprechende Funktionen in Teilprobleme zu unterteilen. Diese Eigenschaften führen dazu, dass man ohne großen Aufwand strukturierten und verständlichen Programmcode erzeugen kann. Sowohl beim Testen, als auch bei der Wiederverwendbarkeit führen diese Eigenschaften zu enormen Vorteilen gegenüber imperativen Programmiersprachen. Teilfunktionen lassen sich einfach wiederverwenden und können unabhängig vom Rest des Programms getestet werden. Die Freiheit von Seiteneffekten führt zudem dazu, dass Fehler im Programmablauf schneller ausfindig gemacht werden und reproduziert werden können, da während des Programmablaufs keine Zustandsveränderungen berücksichtigt werden müssen. Ein weiteres Merkmal von funktionalen Programmiersprachen ist die einfache Syntax. Es reichen grundsätzlich eine Reihe einfacher Konstrukte und wenige Schlüsselwörter aus, um eine funktionale Programmiersprache zu beschreiben. Allein diese Eigenschaften zeigen, warum sich ein Großteil der Forschung mit dem Bereich der funktionalen Programmierung beschäftigt und warum immer mehr Konzepte auch außerhalb von funktionalen Programmiersprachen Anwendung finden. Für diese Arbeit sind vor allem die einfache Syntax, die Auswertungseigenschaften und die mathematischen Eigenschaften im Bezug auf die Gleichheit und Korrektheit von Programmen ein ausschlaggebender Punkt, warum sich funktionale Programmiersprachen für die Untersuchung von Programmoptimierungen sehr gut eignen.

2.1.1 Funktionale Kernsprachen

Das Grundgerüst jeder funktionalen Programmiersprache ist die sogenannte *Kernsprache*, die auch als *Kalkül* bezeichnet wird. Dabei beschreibt der *Kalkül* die Syntax und Semantik der Sprache. Bei der Syntax handelt es sich um die Regeln für den Aufbau eines Ausdrucks und bei der Semantik um die Bedeutung der Ausdrücke.
Der Ursprung der funktionalen Programmierung liegt im *Lambda-Kalkül*, der 1930 von Alonzo Church entwickelt wurde und dessen Aufbau mit folgender

kontextfreien Grammatik beschrieben werden kann [SSS14, S. 14]:

$$\mathbf{Expr} ::= V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}\ \mathbf{Expr})$$

Abbildung 2.1: Kontextfreie Grammatik des Lambda-Kalküls

Das V bezeichnet in diesem Fall eine Variable, die aus einer unbegrenzten Menge an Variablennamen erzeugt wird.

Der Lambda-Binder λ in Kombination mit einer Variablen und einem Ausdruck wird als *Abstraktion* bezeichnet. Eine Abstraktion kann als Funktion verstanden werden, die aus einem Input V besteht und dem Funktionsrumpf, in dem der Input gültig ist. Diese Form von Funktionen gelten als *anonym*, da sich damit namenlose Funktionen darstellen lassen. Ein Beispiel hierfür ist die Identitätsfunktion $id(x)=x$, die sich mit $\lambda x.x$ darstellen lässt.

Zuletzt bezeichnet (**Expr Expr**) eine *Anwendung*, mit der sich Funktionen auf Argumente anwenden lassen. Hierbei spielt es keine Rolle, ob das Argument selbst eine Funktion oder ein beliebig anderer Ausdruck ist. Diese Eigenschaft macht den Lambda-Kalkül zu einem Kalkül *höherer Ordnung*, in dem beispielsweise Funktionen auf sich selbst angewendet werden können. Ein Beispiel hierfür ist der Ausdruck $(\lambda x.x)\ (\lambda x.s)$, in dem die Identitätsfunktion auf eine beliebige andere Funktion $\lambda x.s$ angewendet wird:

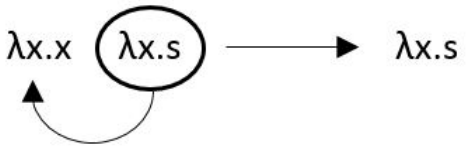

Abbildung 2.2: Verdeutlichung einer Funktionsanwendung

Die Variablen im Lambda-Kalkül lassen sich in *freie* und *gebundene Variablen* unterteilen. Hierfür werden zwei Funktionen definiert, um die Menge der gebundenen oder freien Variablen zu bestimmen. Die Funktion *FV(t)* bestimmt die Menge der freien Variablen für einen Ausdruck t und die Funktion *BV(t)* analog für die gebundenen Variablen. Die Unterteilung der Variablen

in freie und gebundene ist notwendig, um den Gültigkeitsbereich der Variablen zu definieren [SSS14, S. 15]:

$$\begin{array}{lcllcl} FV(x) & = & x & BV(x) & = & \emptyset \\ FV(\lambda x.s) & = & FV(s) \setminus \{x\} & BV(\lambda x.s) & = & BV(s) \cup \{x\} \\ FV(s\ t) & = & FV(s) \cup FV(t) & BV(s\ t) & = & BV(s) \cup BV(t) \end{array}$$

Abbildung 2.3: Bestimmung von freien und gebundenen Variablen mittels Funktionen

2.1.2 Der erweiterte Lambda-Kalkül LR

In dieser Arbeit wird statt dem Lambda-Kalkül der erweiterte Lambda-Kalkül *LR* aus [SSS08] und [SS15] verwendet. Diese Kernsprache ähnelt der Kernsprache von bekannten funktionalen Programmierprachen wie *Haskell* und erweitert den Lambda-Kalkül um folgende Konstrukte [SSS08, S. 510]:

$$\begin{array}{rcl} E & ::= & V \mid (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\mathtt{seq}\ E_1\ E_2) \mid (\mathtt{case}_T\ E\ Alt_1 \ldots Alt_{|T|}) \mid (E_1\ E_2) \\ & & (\lambda\ V.E) \mid (\mathtt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \mathtt{in}\ E) \\ Alt & ::= & (Pat\ \rightarrow\ E) \\ Pat & ::= & (c\ V_1 \ldots V_{\mathrm{ar}(c)}), \end{array}$$

Abbildung 2.4: LR-Kalkül

Das Symbol c bezeichnet einen *Konstruktor* mit der Stelligkeit $ar(c) \geq 0$. Der *case*-Ausdruck beschreibt eine Fallunterscheidung. Dieser besteht aus einem Eingabeausdruck und einer festen Anzahl von *Alternativen*. Jede Alternative besteht aus einem Konstruktor und dem Ausgabeausdruck. Das folgende Beispiel zeigt einen Case-Ausdruck mit zwei Alternativen. Im Falle *Nil* wird *Nil* als Resultat ausgegeben. Sollte der Eingabeausdruck von der Form $c\ x\ e$ sein, so wird das erste Argument des Konstruktors ausgegeben:

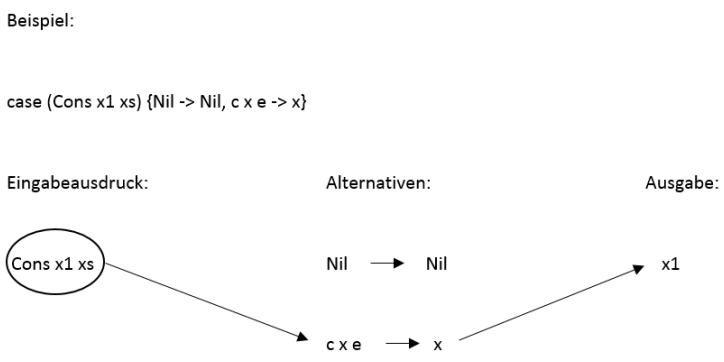

Abbildung 2.5: Vereinfachte Auswertung eines Case-Ausdrucks

Mit dem *seq*-Konstrukt lässt sich eine strikte-Auswertung erzwingen. In diesem Fall muss der Ausdruck E_1 erfolgreich ausgewertet werden bevor der Ausdruck E_2 betrachtet wird.
Ein *letrec*-Ausdruck besteht aus *Variablenbindungen* und einem Ausdruck. Mit Hilfe der Bindungen kann man Variablen Ausdrücke zuweisen. Diese Variablen sind dann im sogenannten *IN*-Ausdruck des *letrec* gültig. Die Variablen, die in den Bindungen definiert werden, müssen jeweils paarweise unterschiedlich sein, da sonst der Ausdruck nicht eindeutig ausgewertet werden kann oder es zu einem Kreisschluss kommt:

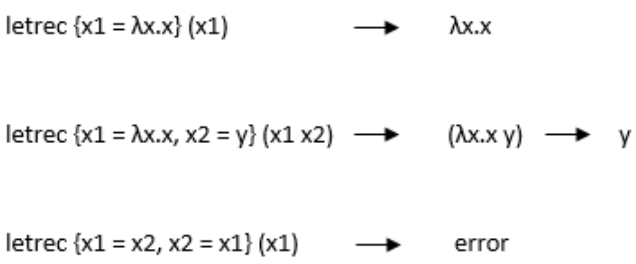

Abbildung 2.6: Vereinfachte Auswertung von Letrec-Beispielen

2.2 Auswertung von funktionalen Programmen

Um ein Programm auswerten zu können, bedarf es einer Semantik, in der die Bedeutung des Programms definiert wird. Diese Arbeit konzentriert sich

hierbei auf *operationale Semantiken* in Form von *small-step Ersetzungssystemen*. Small-step Ersetzungssysteme werten ein Programm aus, in dem sie den Wert in kleinen Schritten durch *Termersetzungen* bestimmen.
Termersetzungen lassen sich mit Hilfe von *Substitutionen* durchführen. Bei einer Substitution $s[t/x]$ werden alle freien Variablen x im Ausdruck s durch den Term t ersetzt. Hierbei kann es zu Namenskonflikten kommen, die später mit Hilfe von Umbenennungsverfahren behandelt werden. In diesem Fall wird angenommen, dass $BV(s) \cap FV(t) = \emptyset$ gilt. Substitutionsregeln für den Lambda-Kalkül können wie folgt definiert werden [SSS14, S. 16]:

$$
\begin{array}{lcl}
x[t/x] & = & t \\
y[t/x] & = & y, \textit{ falls } x \neq y \\
(\lambda y.s)[t/x] & = & \begin{cases} \lambda y.(s[t/x]) & \textit{falls } x \neq y \\ \lambda y.s & \textit{falls } x = y \end{cases} \\
(s_1\ s_2)[t/x] & = & (s_1[t/x]\ s_2[t/x]) \\
(s_1\ s_2)[t/x] & = & (s_1[t/x]\ s_2[t/x])
\end{array}
$$

Abbildung 2.7: Subtitutionsregeln für den Lambda-Kalkül

Der Ausdruck $\lambda x.y[\lambda x.x/y]$ wird mit Hilfe dieser Regeln zum Ausdruck $\lambda x.\lambda x.x$.

Ein weiteres wichtiges Konstrukt bei der Auswertung von Programmen sind *Kontexte*. Kontexte sind Ausdrücke in denen ein Unterausdruck fehlt und durch einen Platzhalten, ein sogenanntes *Loch* $[\cdot]$, repräsentiert wird. An dieser freien Stelle im Ausdruck kann ein beliebiger anderer Ausdruck eingesetzt werden und somit einen neuen Gesamtausdruck erzeugen. Kontexte lassen sich ebenfalls durch eine kontextfreie Grammatik beschreiben [SSS14, S. 16]:

$$\mathbf{C} = [\cdot] \mid \lambda V.\mathbf{C} \mid (\mathbf{C}\ \mathbf{Expr}) \mid (\mathbf{Expr}\ \mathbf{C})$$

Abbildung 2.8: Kontexte im Lambda-Kalkül

Ein Beispiel für Kontexte ist der Ausdruck $C = \lambda x.[\cdot]$, der mit der Eingabe $C[\lambda y.(x\ y)]$ zum Ausdruck $\lambda x.\lambda y.(x\ y)$ wird.

Zwei weitere wichtige Konzepte bei der Reduktion von Ausdrücken sind die α-Umbenennung und die β-Reduktion. Mit der α-Umbennenung wird sichergestellt, dass die *Distict Variable Convention (DVC)* eigehalten wird. Diese sagt aus, dass in einem Ausdruck alle gebundenen Variablen unterschiedliche Namen haben müssen und sich von den freien Variablen unterscheiden müssen. Dieser Umbenennungschritt hat folgende Form [SSS14, S. 17]:

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ falls } y \notin BV(C[\lambda x.s]) \cup FV(C[\lambda x.s])$$

Abbildung 2.9: α-Umbennenungsschritt

Bei der β-Reduktion handelt es sich um eine der wichtigsten Reduktionsregeln. Diese beschreibt die allgemeine Auswertung von Funktionsanwendnungen auf Ausdrücke und ist wie folgt definiert [SSS14, S. 17]:

$$(\beta) \qquad (\lambda x.s)\ t \to s[t/x]$$

Abbildung 2.10: β-Reduktionsschritt

Bei der Auswertung von Ausdrücken unterscheidet man zwischen der *Normalordnungsreduktion (call-by-name oder nicht-strikt)* und der *Anwendungsordnung (call-by-value oder strikte Auswertung)*. Bei der Anwendungsordnung darf eine β-Reduktion nur dann angewendet werden, wenn das Argument der Funktion eine Abstraktion oder Variable ist. Die Normalordnungsreduktion, mit der sich diese Arbeit hauptsächlich beschäftigen wird, reduziert hingegen den Ausdruck an der am weitesten oben und links stehenden Stelle. Die Normalordnungsreduktion kann mit folgender Grammatik für Reduktionskontexte beschrieben werden [SSS14, S. 18]:

$$\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr})$$

Abbildung 2.11: Normalordnungs-Reduktionskontexte für den Lambda-Kalkül

Mit den nun kennengelernten Definitionen lässt sich folgende Aussage treffen:
Wenn $s \xrightarrow{\beta} t$, also s reduziert mittels β zu t, dann ist $R[s] \xrightarrow{no} R[t]$ ein Normalordnungsreduktionsschritt.

2.2.1 Auswertung des LR-Kalküls

Um den LR-Kalkül auswerten zu können bedarfs es einer Reihe an Reduktionsregeln, mit denen man die neu eingeführten Konstrukte auswerten kann [SS15, S. 4]:

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

(cp-in) $(\texttt{letrec}\ x_1 = (\lambda x.s)^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[x_m^{\mathsf{vis}}]) \to (\texttt{letrec}\ x_1 = (\lambda x.s), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \ \texttt{in}\ C[(\lambda x.s)])$

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

(llet-in) $(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^{\mathsf{sub}}) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$

(llet-e) $(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^{\mathsf{sub}}\ \texttt{in}\ r) \to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$

(lapp) $C[((\texttt{letrec}\ Env\ \texttt{in}\ t)^{\mathsf{sub}}\ s)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))]$

(lcase) $C[(\texttt{case}_K\ (\texttt{letrec}\ Env\ \texttt{in}\ t)^{\mathsf{sub}}\ alts)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_K\ t\ alts))]$

(lseq) $C[(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^{\mathsf{sub}}\ t)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))]$

(seq-c) $C[(\texttt{seq}\ v^{\mathsf{sub}}\ t)] \to C[t]$ if v is a value

(seq-in) $(\texttt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{seq}\ x_m^{\mathsf{vis}}\ t)]) \to (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[t])$

(seq-e) $(\texttt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[(\texttt{seq}\ x_m^{\mathsf{vis}}\ t)]\ \texttt{in}\ r) \to (\texttt{letrec}\ x_1 = (c\ \overrightarrow{t}), \{x_i = x_{i-1}\}_{i=2}^{m}, Env, y = C[t]\ \texttt{in}\ r)$

(case-c) $C[(\texttt{case}_K\ (c_i\ \overrightarrow{t})^{\mathsf{sub}}\ \ldots((c_i\ \overrightarrow{y})\ \texttt{->}\ t)\ldots)] \to C[(\texttt{letrec}\ \{y_i = t_i\}_{i=1}^{n}\ \texttt{in}\ t)]$ if $n = ar(c_i) \geq 1$

(case-c) $C[(\texttt{case}_K\ c_i^{\mathsf{sub}}\ \ldots\ (c_i\ \texttt{->}\ t)\ldots)] \to C[t]$ if $ar(c_i) = 0$

(case-in) $\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[\texttt{case}_K\ x_m^{\mathsf{vis}}\ \ldots((c_i\ \overrightarrow{z})\ \texttt{->}\ t)\ldots] \to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), \{y_i = t_i\}_{i=1}^{n}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[(\texttt{letrec}\ \{z_i = y_i\}_{i=1}^{n}\ \texttt{in}\ t)]$
where $n = ar(c_i) \geq 1$ and y_i are fresh variables

(case-in) $\texttt{letrec}\ x_1 = c_i^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ C[\texttt{case}_K\ x_m^{\mathsf{vis}}\ \ldots\ (c_i\ \texttt{->}\ t)\ldots] \to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ \ C[t]$ if $ar(c_i) = 0$

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

(case-e) $\texttt{letrec}\ x_1 = c_i^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, u = C[\texttt{case}_K\ x_m^{\mathsf{vis}}\ \ldots\ (c_i\ \texttt{->}\ r_1)\ldots], Env\ \texttt{in}\ r_2 \to \texttt{letrec}\ x_1 = c_i, \{x_i = x_{i-1}\}_{i=2}^{m}\ldots, u = C[r_1], Env\ \texttt{in}\ r_2$ if $ar(c_i) = 0$

Abbildung 2.12: Reduktionsregeln des LR-Kalküls

Die Markierungen, die in den Reduktionsregeln vorkommen, stammen von einem Algorithmus der entscheidet, welche Stelle des Ausdrucks als nächstes reduziert werden soll. Dieser Algorithmus verwendet dafür *Label*, mit denen er die Ausdrücke und Unterausdrücke markiert.
Das Label *top* steht für die Reduktion eines Gesamtausdrucks.
Das Label *sub* hingegen für die Reduktion eines Unterausdrucks.
Zuletzt markiert *vis* bereits besuchte Unterausdrücke und *nontarg* bereits besuchte Variablen, die nicht das Ziel einer *cp*-Reduktion sind [SS15, S. 4]:

$$
\begin{array}{ll}
(s\ t)^{\mathsf{sub}\vee\mathsf{top}} & \to (s^{\mathsf{sub}}\ t)^{\mathsf{vis}} \\
(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s)^{\mathsf{top}} & \to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ s^{\mathsf{sub}})^{\mathsf{vis}} \\
(\texttt{letrec}\ x = s, \mathit{Env}\ \texttt{in}\ C[x^{\mathsf{sub}}]) & \to (\texttt{letrec}\ x = s^{\mathsf{sub}}, \mathit{Env}\ \texttt{in}\ C[x^{\mathsf{vis}}]) \\
(\texttt{letrec}\ x = s, y = C[x^{\mathsf{sub}}], \mathit{Env}\ \texttt{in}\ t) & \to (\texttt{letrec}\ x = s^{\mathsf{sub}}, y = C[x^{\mathsf{vis}}], \mathit{Env}\ \texttt{in}\ t),\ \text{where } C \text{ is not trivial} \\
(\texttt{letrec}\ x = s, y = x^{\mathsf{sub}}, \mathit{Env}\ \texttt{in}\ t) & \to (\texttt{letrec}\ x = s^{\mathsf{sub}}, y = x^{\mathsf{nontarg}}, \mathit{Env}\ \texttt{in}\ t) \\
(\texttt{seq}\ s\ t)^{\mathsf{sub}\vee\mathsf{top}} & \to (\texttt{seq}\ s^{\mathsf{sub}}\ t)^{\mathsf{vis}} \\
(\texttt{case}\ s\ \texttt{of}\ \mathit{alts})^{\mathsf{sub}\vee\mathsf{top}} & \to (\texttt{case}\ s^{\mathsf{sub}}\ \texttt{of}\ \mathit{alts})^{\mathsf{vis}} \\
\texttt{letrec}\ x = s^{\mathsf{vis}\vee\mathsf{nontarg}}, y = C[x^{\mathsf{sub}}], \mathit{Env}\ \texttt{in}\ t & \to \texttt{Fail} \\
(\texttt{letrec}\ x = C[x^{\mathsf{sub}}], \mathit{Env}\ \texttt{in}\ s) & \to \texttt{Fail}
\end{array}
$$

Abbildung 2.13: Der *Labeling Algorithmus*

Die Reduktionsregelns des LR-Kalküls lassen sich in folgende Gruppen einteilen. Die erste Regel beschreibt hierbei die bereits bekannte β-Reduktion, die an das *letrec*-Konstrukt angepasst werden musste.

Bei den *cp*-Reduktionen handelt es sich um zwei Varianten für das Kopieren einer Abstraktion. Die erste Variante *cp-in* kopiert die Abstraktion an die passende Stelle des *in*-Ausdrucks. Hierbei muss man beachten, dass es eine beliebige Stelle in einem Kontext C sein kann. Die zweite Variante kopiert eine Abstraktion in die Bindungen selbst. Das heißt $\{x_i = x_{i-1}\}$ beschreibt eine Verkettung von Variablenbindungen die am Ende dazu führt, dass die Variable x_m (die am Ende y zugewiesen wird) den Wert von x_1 zugewiesen bekommt.

Die Regeln *llet-in* und *llet-e* lösen verschachtelte *letrec*-Ausdrücke auf und fassen diese zu einem *letrec* zusammen. Dafür werden in *llet-in* die Bindungen des obersten *letrec* mit den Bindungen des *letrec* im *in*-Ausdrucks zusammengefasst. In der *llet-e*-Regel werden die Bindungen des obersten *letrec* mit den Bindungen des *letrec* in der Variablenbindung x zusammengefasst und die Variable x bekommt den *in*-Ausdruck des inneren *letrec* zugewiesen.

Mit der *lapp*-Regel wird eine Anwendung in den *in*-Ausdruck des *letrec* reingezogen. Im Allgemeinen lässt sich sagen, dass in den *l...*-Regeln ein *letrec*-Ausdruck an die oberste Stelle gezogen wird. Dies kann man in den folgenden Regeln erneut erkennen.

In der *lcase*-Regel wird ein *case*-Ausdruck aufgelöst, der als Eingabe ein *letrec*-Ausdruck hat. Der *in*-Ausdruck des *letrec* kann hierbei einfach als Eingabeausdruck des *case* übernommen werden und das *letrec* kann herausgezogen werden. Analog verhält sich die *lseq*-Regel.

Die *seq-c*, *seq-in* und *seq-e* Regeln beschreiben den Fall, dass ein *seq*-Ausdruck einen *Wert* als erstes Argument bekommt. Das zweite Argument des *seq*-Ausdrucks kann also ausgewertet werden.

Im LR-Kalkül endet die Auswertung eines Ausdrucks mit einem *Wert*. Dieser Wert ist entweder eine Abstraktion $\lambda x.s$, eine Konstruktoranwendung $c\ \overrightarrow{t}$, ein *Letrec*-Ausdruck der Form $(\texttt{letrec}\ x_1 = (c\ \overrightarrow{t})^{\mathsf{sub}}, \{x_i = x_{i-1}\}_{i=2}^{m}, Env\ \texttt{in}\ x_m)$ oder $(\texttt{letrec}\ Env\ \texttt{in}\ v)$[1]. Diese endgültige Form wird auch *Weak Head Normal Form*, kurz *WHNF*, genannt.

Die Regel *seq-c* beschreibt den Fall, dass ein einfacher *seq*-Ausdruck im ersten Argument zu einer *WHNF* terminiert. Es wird also mit dem zweiten Argument weitergemacht. Die *seq-in*-Regel beschreibt den Fall, dass der *seq*-Ausdruck in einem *letrec*-Ausdruck ist und das erste Argument mit einem Konstruktor belegt wird. Hier wird der *in*-Ausdruck des *letrec* mit dem zweiten Argument des *seq* belegt. Im *seq-e*-Fall befindet sich das *seq* in den Variablenbindungen. Auch hier wird analog das *seq* aufgelöst und es bleibt das zweite Argument.

Die Regel *case-c* beschreibt den Fall, dass ein *case*-Ausdruck als Eingabe einen Konstruktor bekommt. Dabei unterscheidet man zwischen einem Konstruktor mit Stelligkeit 0 oder >0. Hat der Konstruktor keine Argumente, so wird die Reduktion mit dem Ausdruck der passenden *case*-Alternative fortgesetzt. Hat der Konstruktor Argumente, so wird mit Hilfe eines *letrec* der Ausdruck angepasst. Die Konstruktorargumente werden entsprechend der *case*-Alternative den richtigen Variablennamen zugewiesen und der *in*-Ausdruck des *letrec* wird zum Ausdruck der passenden *case*-Alternative.

Die letzten Fälle *case-in* und *case-e* sind analog aufgebaut. Auch hier wird generell unterschieden zwischen einem Konstruktor ohne und mit Argumente. Zudem befinden sich die *case*-Ausdrücke diesmal in einem *letrec*. Im *case-in*-Fall befindet sich der *case*-Ausdruck im *in*-Ausdruck des *letrec* und beim *case-e*-Fall im den Bindungen des *letrec*. In beiden Fällen löst man das *case* auf in dem man den Konstruktorargumenten neuen Variablen zuweist und mit Hilfe des *letrec* dem Ausdruck der passsenden *case*-Alternative zuordnet.

2.3 Analyse von funktionalen Programmen

Bei der Analyse von funktionalen Programmen geht es um die Transformation und Optimierung von Code. Programmtransformationen sind

[1] v ist ein Wert

binäre Relationen die aus einem Ausdruck einen syntaktisch veränderten Ausdruck machen. Für zwei Ausdrücke s und t gilt: $s \xrightarrow{P} t$, P ist eine Programmtransformation.
Beim Anwenden einer Programmtransformation müssen immer die jeweiligen Kontexte betrachtet werden, so ist eine Programmtransformation nur dann vollständig, wenn die Kontexte definiert werden für die diese Programmtransformation gilt. Man schreibt auch $C[s] \xrightarrow{X,P} C[t]$ für $C \in X$ und X ist eine Menge an Kontexten.
Eine Programmtransformation allein macht in diesem Zusammenhang noch keine bedeutende Aussage. Eine Programmtransformation wird dann interessant, wenn die Semantik des Programms unverändert bleibt. In diesem Fall kann man durch eine Programmtransformation die Syntax des Programms verändern, also Reduzieren oder die Auswertung vereinfachen, ohne dabei die Bedeutung zu verändern. Das Programm sagt das gleiche aus, aber verwendet dafür weniger auswertungsintesive Konstrukte oder benötigt allgemein weniger Reduktionsschritte.
Um die semantische Äquivalenz von zwei Ausdrücken in einem beliebigen Kontext zu zeigen, wird der Begriff der kontextuellen Äquivalenz definiert:

Zwei Ausdrücke sind kontextuell Äquivalent $s \sim_c t$, wenn sich beide Ausdrücke in allen Kontexten gleich verhalten. Die Ersetzung eines Programms durch ein zweites Programm kann im Kontext von Außen nicht erkannt werden.

Kurz: $C[s] \downarrow \leftrightarrow C[t] \downarrow$[2].

Programmtransformationen, die diese Eigenschaft erfüllen, heißen *korrekt*. Die vorgestellten Reduktionsregeln des LR-Kalküls sind korrekte Programmtransformationen.

Die Eigenschaft der *Korrektheit* sagt noch nichts über die Laufzeit, die Speicherkomplexität oder das Auswertungsverhalten des resultierenden Ausdrucks aus. Um die Auswertungskomplexität und Zeitkomplexität eines funktionalen Ausdrucks in einem small-step Ersetzungssystem zu messen ist es notwendig, die Reduktionsschritte zu zählen und die entscheidenden Transformationen ausfindig zu machen [SS15, S. 7]:

Sei $\mathfrak{A} :=$ *(lbeta),(case),(seq) und* $\emptyset \neq A \subseteq \mathfrak{A}$. *Sei* t *ein geschlossener Ausdruck*[3] *mit* $t \downarrow t_0$. *Dann ist* rln_A *die Anzahl an a-Reduktionen mit* $a \in$

[2] $\downarrow$ bedeutet das ein Ausdruck mit Hilfe einer Normalordnungsreduktion zu einer WHNF konvergiert und $\uparrow$ das der Ausdruck divergiert

[3] Ausdrücke t die FV(t) = $\emptyset$ erfüllen nennt man *geschlossen*. Ausdrücke für die diese Eigenschaft nicht gilt nennt man *offen*

A. Mit rlnall(t) wird die Anzahl aller Reduktionen in $t \downarrow t_0$ bezeichnet. Im Fall das ein Ausdruck divergiert, wird die Anzahl ∞ definiert.

Der entscheidende Indikator für eine Optimierung ist der *rln*-Wert. Dieser zählt nur die *lbeta-*, *case-* und *seq*-Reduktionsregeln. Dabei fasst der Begriff *case*-Regel die Regeln *case-c*, *case-in* und *case-e* zusammen. Der Begriff *seq*-Regeln die Regeln *seq-c*, *seq-in* und *seq-e*. Hierbei können die *cp*-Regeln vernachlässigt werden, da diese meist der letzte Reduktionschritt sind, oder nach jeder *cp*-Reduktion eine *lbeta-* oder *seq*-Reduktion folgt. Die Reduktionsregeln *lapp*, *lcase*, *lseq* und alle *let*-Regeln können ebenfalls vernachlässigt werden, da diese effizienter auf einer abstrakten Maschine implementiert werden können. [SS15, S.8]

Schließlich kann nun der Begriff der Optimierung (engl. Improvement) eingeführt werden. Hierfür wird eine Relation eingeführt die aussagt, ob ein Ausdruck die Optimierung eines zweiten Ausdrucks ist [SS15, S.8]:

Für $s,t \in A \subseteq \mathfrak{A}$, sei $s \preceq_A t$ (t wir A-optimiert von s), also $s \sim_c t$ und für alle Kontexte C gilt: Wenn C[s], C[t] sind geschlossen, dann gilt $rln_A(C[s]) \leq rln_A(C[t])$.
Wir schreiben $t \succeq_A s$, wenn $s \preceq_A t$ gilt. Wenn $s \preceq_A t$ und $s \succeq_A t$, dann schreiben wir $s \approx_A t$. Eine Programmtransformation P ist eine A-Optimierung, wenn $P \subseteq \succeq_A$ gilt.

Kapitel 3

Konzeption

In diesem Kapitel wird mit Hilfe der Grundlagen, die in Kapitel 2 eingeführt wurden, ein Werkzeug zur Überprüfung von Programmoptimierungen in funktionalen Programmiersprachen entwickelt. In Abschnitt 3.1 werden die Anforderungen an das Werkzeug formuliert und die zur Erfüllung dieser benötigten Bestandteile in 3.2 erarbeitet. Die konkrete Implementierung wird dann im Kapitel 4 erläutert und detailliert beschrieben.

3.1 Anforderungsanalyse

Um ein Werkzeug implementieren zu können, das dem Ziel dieser Arbeit gerecht wird, ist es notwendig genaue Anforderungen zu formulieren, mit denen ein Rahmen für die Implementierung gebildet wird.
In Kapitel 2 haben wir bereits die Kernsprache *LR* in Abbildung 2.4 und die dazugehörigen Reduktionsregeln in Abbildung 2.12 kennengelernt.

Die erste Anforderung an das Werkzeug ist also die Durchführung einer *Normalordnungsreduktion* eines *LR-Ausdrucks*. Neben der Reduktion von *LR-Ausdrücken* sollen diese auch mit Hilfe eines *Parsers* geparst werden können. Hierbei ist es wichtig, dass die Auswertung des Ausdrucks analysierbar sein muss. Der Ausdruck muss also schrittweise reduziert werden und während jedes einzelnen Schritts muss die angewendete Reduktionsregel erkennbar sein. Nur so lässt sich später mit Hilfe des *rln*-Wertes aussagen, ob es sich bei einer Programmtransformation um eine Optimierung handeln kann, oder ob zwei Ausdrücke $s \preceq_A t$ erfüllen.
Nachdem das Fundament des Werkzeugs definiert wurde, müssen nun Funktionen bereitgestellt werden die *rln* und *rlnall* berechnen. Diese zwei Werte bilden der Kern der Analyse von Programmoptimierungen.

Da der *Parser* die Eingabe von eigenen *LR-Ausdrücken* ermöglicht, sollte es auch möglich sein eigene Programmtransformationen neben den

Normalordnungsregeln des *LR-Kalküls* anzugeben und analysieren zu können.

Die Analyse der Ausdrücke und Transformationen soll in Form von detaillierten Berichten in einem gängigen Textformat visualisiert werden.

Zuletzt soll eine einfache graphische Oberfläche implementiert werden, die dem Nutzer eine benutzerfreundliche Alternative zu der direkten Nutzung der oben genannten Funktionen bietet. Diese soll einfache Funktionen eines Texteditors bereitstellen und elementare Funktionen, wie zum Beispiel die Berechnung des *RLN*-Wertes, bereitstellen.

3.2 Konzept

Nachdem die Anforderungen an das Werkzeug formuliert wurden müssen nun die einzelnen Bestandteile des Werkzeugs erarbeitet und für die Umsetzung relevante Hilfsmittel vorgestellt werden. Zu Beginn wird kurz auf die für die Implementierung verwendete funktionale Programmiersprache *Haskell* eingegangen. Im Anschluss werden dann die einzelnen verwendeten Hilfsmittel präsentiert, die zur Umsetzung der vorher definierten Anforderungen beitragen.

3.2.1 Haskell

Die funktionale Programmiersprache *Haskell*[1] ist einer der bekanntesten funktionalen Programmiersprachen die weltweit genutzt wird. Compiliert wird Haskell mit Hilfe des *Glasgow Haskell Compiler*[2].

Die Syntax und Semantik von Haskell ähnelt der des bereits eingeführten *LR-Kalküls* und bietet sich deshalb zur Implementierung des Werkzeugs an. Haskell wird vollständig durch den Haskell Language Report[3] beschrieben und zeichnet sich durch seine *verzögerte Auswertung*[4] aus, bei der ein Ausdruck nur dann ausgewertet wird, wenn das Ergebnis benötigt wird. In diesem Abschnitt soll nur ein kurzer Einblick in die Funktionsweise von Haskell gegeben werden.

[1] Siehe `https://www.haskell.org/`
[2] Siehe `https://www.haskell.org/ghc/`
[3] Siehe [Mar10]
[4] auch *nicht-strikte Auswertung* genannt oder engl. *Lazy-Evaluation*

Haskell ist eine *reine* funktionale Programmiersprache und hat ein *polymorphes*, *statisches* und *starkes* Typsystem. Mit polymorphen Typsystemen kann man *schematische Typen* verwenden und Typen anhand von *Typvariablen* definieren. Statische Typsysteme berechnen den Typ nicht zur Laufzeit, so dass es während der Laufzeit nicht mehr zu Typfehlern kommen kann. Durch die starke Typisierung muss jeder Ausdruck ordentlich getypt sein. Eine Ausnahme bildet hierbei die Herleitung von Typen anhand vorhandener getypter Ausdrücke, die allgemein als *Typinferenz* bezeichnet wird. Ein Beispiel für verschiedene Typdeklarationen sieht wie folgt aus [HF92]:

```
    5  :: Integer
  'a'  :: Char
  inc  :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

Abbildung 3.1: Typdeklaration in Haskell

Funktionen wie *inc* werden mit einem → Symbol deklariert. In diesem Fall hat die Funktion *inc* als Input einen Wert vom Typ *Integer* und liefert als Resultat einen Wert vom Typ *Integer*. Listen werden mit eckigen Klammern oder mit dem Konstruktor **:** deklariert. Ein Beispiel für eine Liste ist **(a:[])**. Hier wird eine Liste mit dem Element *a* definiert. Die nächste Abbildung veranschaulicht die beiden Deklarationsformen für Listen nochmal genauer:

['a','a','a'] ('a':('a':('a':[])))

Abbildung 3.2: Listen in Haskell

Mit Modulen kann man in Haskell Programmteile kapseln und modularisieren. Sie legen neben den Namensraum auch fest, welche Funktionen beim Importieren des Moduls zur Verfügung stehen und können abstrakte Datentypen definieren.

Ein Modul besteht aus den importieren Modulen die es verwendet, Typdefinitionen, Datentypdefinitionen und Funktionen. Importe müssen immer am Anfang eines Moduls deklariert werden, der Rest kann frei angeordnet werden [HF92]:

```
module Tree ( Tree(Leaf,Branch), fringe ) where

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

fringe :: Tree a -> [a]
fringe (Leaf x)             = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Abbildung 3.3: Ein Modul in Haskell

In diesem Modul wird eine Baumstruktur beschrieben, die aus dem Datentyp *Tree* und der Funktion *fringe* besteht. Das Modul exportiert sowohl den Datentyp, als auch die Funktion. Die Funktion *fringe* gibt eine Liste aller Elemente zurück, die in den Blättern der Baumstruktur stehen.

Das Modul kann jetzt von außerhalb importiert werden und es können beispielsweise die Blatt-Elemente ausgegeben werden [HF92]:

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

Abbildung 3.4: Beispiel: Modulimport

Ein weiteres wichtiges Konstrukt in Haskell sind *Monaden*:

Monadisches Programmieren ist (aus Programmierersicht) eine bestimmte Strukturierungsmethode, um sequentiell ablaufende Programme in einer funktionalen Programmiersprache zu implementieren. Der Begriff Monade stammt aus dem Teilgebiet der Kategorientheorie der Mathematik. Ein Typkonstruktor ist eine Monade, wenn er etwas verpackt und bestimmte

Operationen auf dem Datentyp zulässt, wobei die Operationen die sog. monadischen Gesetze erfüllen müssen.[SSS14, S. 182]

Dieses Konstrukt ist notwendig, um mit einem Haskellprogramm den Zustand des Rechners verändern zu können. In dieser Arbeit werden Monaden genutzt, um Lese- und Schreibprozesse, sowie Ein- und Ausgabeoperationen zu realisieren. Die Typklasse *Monad* hat dabei folgende Struktur:

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

Mit Typklassen wird in Haskell *ad hoc Polymorphismus* umgesetzt. *Ad hoc Polymorphismus* beschreibt den Fall, dass Funktionen mehrfach implementiert werden und sich beim Anwenden auf unterschiedliche Typen anders verhalten. Die Funktionen werden hierbei *Überladen*. Ein Beispiel hierfür ist der Additionsoperator *'+'*, der sich bei der Addition von *Integer* anders verhalten muss, als bei *Double*. [SSS14, S. 114]

Die monadischen Operatoren verhalten sich wie folgt:

- *bind* / >>= - Kombiniert zwei monadische Werte, wobei der erste Wert von der zweiten Aktion genutzt wird
- *then* / >> - Kombiniert zwei monadische Werte, wobei der erste Wert nicht von der zweiten Aktion benötigt wird
- *return* - Hebt einen Wert in die Monade (lift)
- *fail* - Fehler

Eine Instanz der Typklasse *Monad*, die in dieser Arbeit genutzt wird, ist der Datentyp *Maybe*. Mit diesem lässt sich die Ausgabe einer Funktion aufteilen in:

- **Just a** - Die Funktion konnte einen Wert vom Typ a liefern
- **Nothing** - Die Funktion konnte keinen Wert liefern

3.2.2 Lexikalische-, syntaktische- und semantische Analyse

Bevor Programmcode analysiert werden kann, muss zunächst die Grammatik der Sprache definiert werden. Diese lässt sich in Haskell mit Hilfe des *data*-Konstruktors leicht als *Datentyp* implementieren.

Die Analyse von Programmcode besteht aus der *lexikalischen*, *syntaktischen* und *semantischen* Analyse der Eingabe. Diese drei Schritte werden von einem *Parser* durchgeführt, der die Korrektheit der Eingabe anhand einer kontextfreien Grammatik überprüft.

Die lexikalische Analyse wird von einem *Tokenizer*[5] durchgeführt. Dieser fasst die Eingabe in logisch zusammenhängende Zeichen[6] zusammen und entfernt unnötige Symbole wie Programmkommentare oder Leerzeichen. An dieser Stelle kann bereits eine Eingabe als ungültig eingestuft werden, wenn sie beispielsweise unerlaubte Symbole enthält.

Die nächsten Schritte sind die syntaktische und semantische Analyse. Diese Schritte werden vom eigentlichen Parser durchgeführt. Hier werden die Symbolgruppen in einen Syntaxbaum überführt und es wird kontrolliert, ob dieser Syntaxbaum der angegebenen Grammatik der Sprache entspricht. Die syntaktische Analyse konzentriert sich hierbei auf die Anordnung der Symbolgruppen gemäß der Grammatik, während bei der semantischen Analyse unter anderem geprüft wird, ob genutzte Variablen tatsächlich vorher deklariert worden sind.

Es gibt verschiedene Methoden um Parser zu implementieren. Eine davon ist die Verwendung von *Parsergeneratoren*. In dieser Arbeit wird der Parsergenerator *Happy*[7] verwendet.

Happy ist ein Parsergenerator, der Anhand einer kontextfreien Grammatik einen Shift-Reduce-Parser in Form eines Haskellmoduls erstellt. Shift-Reduce-Parser sind LR-Parser die eine Eingabe von links nach rechts verarbeiten und dadurch eine Rechtsherleitung erzeugen.[8] Im folgenden Abschnitt werden grundlegende Erklärungen aus [Sch] wiedergegeben.

[5] auch *Lexer* genannt
[6] engl. *Token*
[7] Siehe [Mar]
[8] L steht für links-rechts und R für Rechtsherleitung.

Die Syntax der Eingabe wird dabei deterministisch geparst, das heißt der Parser kann nicht zurückgesetzt werden, und es handelt sich bei dieser Methode um eine *Bottom-Up*-Syntaxanalyse. Im folgenden Beispiel soll eine Rechtsherleitung für einen LR-Ausdruck verdeutlicht werden:

Gegeben sei die bereits bekannte kontextfreie Grammatik des LR-Kalküls:

$$
\begin{array}{rcl}
E & ::= & V \mid (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\mathtt{seq}\ E_1\ E_2) \mid (\mathtt{case}_T\ E\ Alt_1 \ldots Alt_{|T|}) \mid (E_1\ E_2) \\
 & & (\lambda\ V.E) \mid (\mathtt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \mathtt{in}\ E) \\
Alt & ::= & (Pat\ \rightarrow\ E) \\
Pat & ::= & (c\ V_1 \ldots V_{\mathrm{ar}(c)}),
\end{array}
$$

und folgender Ausdruck:

(λx.s (s t))

Eine Rechtsherleitung sieht dann wie folgt aus[9]:

$$
\begin{array}{ll}
\mathrm{E} & \rightarrow (\mathrm{E}\ \underline{\mathrm{E}}) \\
 & \rightarrow (\mathrm{E}\ (\mathrm{E}\ \underline{\mathrm{E}})) \\
 & \rightarrow (\mathrm{E}\ (\underline{\mathrm{E}}\ \mathrm{t}) \\
 & \rightarrow (\underline{\mathrm{E}}\ (\mathrm{s}\ \mathrm{t}) \\
 & \rightarrow (\lambda\mathrm{x}.\underline{\mathrm{E}}\ (\mathrm{s}\ \mathrm{t}) \\
 & \rightarrow (\lambda\mathrm{x}.\mathrm{s}\ (\mathrm{s}\ \mathrm{t})
\end{array}
$$

Der Parser in Happy wird mittels *Parserspezifikation* definiert, die im allgemeinen aus einer Grammatik besteht. Der dadurch entstandene Parser beinhaltet nicht die lexikalische Analyse. Diese muss separat implementiert werden. Für diese Arbeit muss also eine Parserspezifikation erstellt werden,

[9]Das nächste Symbol das bearbeitet wird ist unterstrichen

aus der Happy das eigentliche Parsermodul generiert. Zusätzlich muss ein separates Modul implementiert werden, mit dem die lexikalische Analyse durchgeführt wird und dessen Ausgabe dann vom Parser bearbeitet wird. Im Praktikum *Funktionale Programmierung* [Sab15] wurde bereits ein Lexer und Parser vorgestellt, der sich zur Umsetzung dieses Werkzeugs sehr gut eignet.

Zusammenfassend müssen folgende Bestandteile implementiert werden:

- Ein Haskellmodul, dass die Grammtik des LR-Kalküls als Datentyp implementiert
- Ein Haskellmodul, dass die lexikalische Analyse implementiert
- Ein Haskellmodul, dass den Parser implementiert. Dieses Modul kann mittels Parserspezifikation durch Happy generiert werden.

3.2.3 Auswertung und Analyse des LR-Kalküls

In [Eid15] wurde bereits der Versuch unternommen eine automatische Normalordnungsreduktion von vereinfachten LR-Ausdrücken durchzuführen. Hier wurden bereits Teile der Reduktion implementiert und die Berechnung der *RLN*-Werte durchgeführt. Diese Arbeit kann also als Ansatz genommen werden, um die vollständige Normalordnungsreduktion zu implementieren und um die Anforderungen an diese Arbeit zu erfüllen. Dabei müssen folgende Punkte ergänzt werden:

- Der Parser und die vollständige LR-Grammatik. Siehe 3.2.2
- Die fehlenden Reduktionsregeln der Normalordnungreduktion, dazu gehören:
  - lcase
  - lseq
  - seq-c
  - seq-in
  - seq-e
  - case-c
  - case-in

  - case-e

- Eine detaillierte Ausgabe der Auswertung / Visualisierung der Ergebnisse
- Die Möglichkeit eigene Programmtransformationen zu definieren
- Eine graphische Benutzerschnittstelle zur einfachen Bedienung

Sobald die vollständige Grammatik des LR-Kalküls implementiert ist, können die fehlenden Reduktionsregeln ergänzt werden. Hierbei muss sichergestellt werden, dass jede Regel eindeutig identifizierbar ist, damit diese bei der Auswertung analysiert werden kann. Dazu kann ein eigener Datentyp implementiert werden, der bei jedem Reduktionsschritt aussagt, um welche Regeln es sich handelt oder ob der Ausdruck bereits eine WHNF ist.
Bei der Analyse und Visualisierung der Ergebnisse müssen folgende Informationen bereitgestellt werden:

- RLN- und RLNALL-Werte
- Der Ausdruck vor und nach der Umbenennung
- Die vollständige Reduktion des Ausdrucks mit den einzelnen Schritten und der verwendeten Regeln
- Der resultierende Ausdruck nach der Auswertung

Die Möglichkeit eigene Programmtransformationen angeben zu können, sollte in einem eigenen Modul bereitgestellt werden, da hier die Anzahl der kommenden Funktionen unbekannt ist. Als Vorlage kann hier eine der bekannten zusätzlichen Programmtransformationen, wie die *Garbage collection*[10], implementiert werden.

3.2.4 Tests

Zum Testen der Kernfunktionen des Werkzeugs bietet es sich an ein eigenes Testmodul bereitzustellen, in dem die Testfunktionen für die einzelnen Komponenten definiert werden. Diese Funktionen sollen einen möglichst großen Bereich des Werkzeugs abdecken. Hierbei können die einzelnen Reduktionsregeln separat getestet werden. Diese Form von Tests ist auch als *Komponententest*[11] bekannt. Die graphische Benutzerschnittstelle selbst, sowie die

[10] Siehe *extra transformation rules* [SS15]
[11] engl. Unit test

Analyse- und Auswertungsausgaben müssen hierbei manuell getestet werden. Die manuellen Tests können hierbei als *Akzeptanztests*[12] verstanden werden, bei dem der Entwickler gleichzeitig die Rolle des Benutzers einnimmt und das Werkzeug auf seine Funktionalität untersucht.

3.2.5 Graphische Benutzeroberfläche

In [Cas13] wurde bereits eine graphische Benutzerschnittstelle mit Haskell zu einem verwandten Thema implementiert. Da die Anforderungen an die Benutzerschnittstelle ähnlich sind kann hier ein Großteil der Implementierung übernommen werden. Einzig die Ausgaben müssen jeweils durch die neuen Funktionen ersetzt werden.

[12] engl. User Acceptance Tests (UAT)

Kapitel 4

Implementierung

In diesem Kapitel werden nun die Ideen und Konzepte aus Kapitel 3 umgesetzt. Es werden die einzelnen Bestandteile des Werkzeugs implementiert und detailliert vorgestellt.

4.1 LR-Kalkül, Lexer und Parser

In diesem Abschnitt wir der erste Teil des Werkzeugs implementiert. Dazu gehört die Grammatik des LR-Kalküls selbst, die lexikalische Analyse und der Parser. Mit diesem lassen sich Zeichenketten, die einen LR-Ausdruck beschreiben, in den entsprechenden Datentyp in Haskell überführen.

4.1.1 LR-Kalkül

Die Grammatik des LR-Kalkül wird in einem eigenen Modul *Language* in der Datei *Language.hs* implementiert und besteht aus drei Elementen:

Datentypdefinition:

```
-- | Datatype definitions (monomorph)
data TDef tname cname = TDef tname [(cname, Type tname)]
  deriving (Eq, Show)
```

Typdefinition:

```
-- | Type definitions (monomorph)
data Type tname =
    -- ^ Typeconstructor, e.g. ListInt  = TC "ListInt"
    TC tname
    -- ^ Functiontypes
    | (Type tname) :->: (Type tname)
    -- ^ Variables
    | TVar String
 deriving(Ord,Eq,Show)
```

Grammatik des LR-Kalküls:

```
data Expr label cname v =
    -- ^ Variables: x = Var "x"
    Var v
    -- ^ Abstraction: \x -> e = Lam "x" e
    | Lam v (Expr label cname v)
    -- ^ Application: (e1 e2) = App e1 e2
    | App (Expr label cname v) (Expr label cname v)
    -- ^ seq: seq e1 e2 = Seq e1 e2
    | Seq (Expr label cname v) (Expr label cname v)
    -- ^ constructor-application: (c e1 ... en) = Cons ("c") [e1,...,en]
    | Cons cname [Expr label cname v]
    -- ^ Case: case e of {Alt1;...;Altn} = Case e [Alt1,...,Altn]
    | Case (Expr label cname v) [CAlt label cname v]
    -- ^ Letrec: letrec x1=e1;...;xn=en in e
    | Letrec [Binding label cname v] (Expr label cname v)
    -- ^ Label: For labeling an expression
    | Label label (Expr label cname v)
 deriving(Eq,Show)

data CAlt label cname v =
    -- ^ Case-Alternatives: (c x1 ... xn) -> e
= CAlt "c" [x1,...,xn] e
    CAlt cname [v] (Expr label cname v)
 deriving(Eq,Show)

data Binding label cname v =
    -- letrec-bindings: x=e  =  x :=: e
    v :=: (Expr label cname v)
 deriving(Eq,Show)
```

Hierbei wurden die *Case-Alternativen* und *Letrec-Bindungen* separat von der Hauptgrammatik implementiert, um den Code übersichtlicher zu machen. Die Grammatik des LR-Kalkül 2.4 selbst konnte einfach als entsprechende Datenstruktur implementiert werden. Die Datentyp- und Typdefinitionen sind nötig, damit der Parser mit Datentypen in eigenen LR-Programmen umgehen kann. Hierbei beschränkt sich die Grammatik auf monomorphe Typen.

4.1.2 Lexer

Da jetzt die Datenstruktur für den LR-Kalkül bekannt ist, wird nun der Lexer und Parser benötigt, der eine einfache Zeichenkette fehlerfrei in diesen Datentyp überführt. Zu Beginn muss mit einem Lexer sichergestellt werden, dass keine unerlaubten Symbole verwendet wurden und das der Quellcode fehlerfrei in eine Liste von *Token* umgewandelt werden kann, die dann weiter an den Parser gereicht wird. Diese Aufgabe wird vom Modul *Lexer* übernommen, das in der Datei *Lexer.hs* implementiert wird. Anfangs wird der Datentyp *TokenType* definiert, der aus dem eigentlichen Token und der Position des Elements im Quelltext besteht:

```
-- | A Token with its label
type TokenType = Token SourceMark
-- | (Row, Col)
type SourceMark = (Int, Int)
```

Das Zeichen selbst wird hier mit dem Datentyp *Token* implementiert, das aus folgenden Elementen besteht:

```
-- Token-Datatype
data Token label =
    TokenKeyword   String label
    | TokenSymbol    String label
    | TokenVar       String label
    | TokenCons      String label
    | TokenUnknown   Char label
 deriving (Eq, Show)
```

TokenKeyword steht dabei für folgende Schlüsselwörter:

- expression
- data
- case
- of
- letrec
- in
- seq

TokenSymbol für folgende Sonderzeichen:

- \\
- >>=
- ;
- =
- ->
- |
- (
-)
- {
- }

Die restlichen Elemente *TokenVar*, *TokenCons* und *TokenUnknown* stehen für *Variablen*, *Konstruktoren* und unbekannte Symbole. Die Hauptfunktion des Lexer ist *lexProgram*. Diese Funktion überführt den Quellcode in Form einer Zeichenkette in eine Liste von Token. Dabei wird alles aus dem Quellcode entfernt, was nicht zum Programm selbst gehört wie Kommentare, Leerzeichen und andere Textformatierungselemente:

```
lexProgram :: String -> [TokenType]
lexProgram input = lexer input 1 1
lexer []    _ _ = []
lexer ('-':'-':is) z s = lexer (dropWhile (/= '\n') is) (z) 1
lexer input@(i:is) row col
   | isJust lk = case lk of
                    Just (token, offset, rest)
                       -> token:(lexer rest row (col+offset))
                    Nothing -> error "impossible"
   | isUpper i =
       let (tk,rest) = break (not.isAlpha) input
           offset    = length tk
        in TokenCons tk (row,col):(lexer rest row (col+offset))
   | isLower i =
       let (tk,rest) = break (not.isAlphaNum) input
           offset    = length tk
        in TokenVar tk (row,col):(lexer rest row (col+offset))
   | i == '\n' = lexer is (row +1) 1
   | i == '\t' = lexer is (row) (col+8)
   | isSpace i = lexer is row (col + 1)
   | otherwise = (TokenUnknown i (row,col)):lexer is (row) (col +1)
  where lk = (lookupPrefix input keywordsAndToken row col)


lookupPrefix input []    z s = Nothing
lookupPrefix input ((k,r):xs) z s
 | k `isPrefixOf` input = let rest =(drop (length k) input)
                          in Just (r (z,s),length k,rest)
 | otherwise            = lookupPrefix input xs z s
```

Die Funktion geht dabei wie folgt vor:

1. Wenn der Quellcode leer ist, wird eine leere Liste zurückgegeben.
2. Wenn die aktuelle Zeile mit −− beginnt, dann handelt es sich um einen Kommentar und die gesamte Zeile, bis zum Zeilenende \n, wird verworfen
3. Im restlichen Teil der Funktion wird geprüft, um welches Symbol es sich handelt und dieses wird in den Datentyp *TokenType* umgewandelt. Dafür wird es in die richtige Tokenkategorie eingeteilt und wird mit der richtigen Positionsmarkierung versehen. Diese besteht aus der Zeile und Spalte im Quellcode. Die *lookupPrefix* Funktion nimmt dabei den aktuellen Quellcode und sucht das nächste bekannte Symbol aus der Liste aller erlaubten Symbole *keywordsAndToken*[1], dass als Prefix zum aktuellen Quellcode passt. Die entsprechende Tokenkategorie zusammen mit der Position im Text, wird dann zusammen mit dem restlichen Quellcode, ohne erkanntem Prefix, und der Länge des Prefixes zurückgegeben. Konstruktoren werden dabei durch einen Groß-

[1]Eine Liste aller erlaubten Symbole und Schlüsselwörter in der Form: (ZEICHEN,TokenSymbol), wenn es sich um ein Symbol handelt und (ZEICHEN,TokenKeyword), wenn es sich um ein Schlüsselwort handelt

buchstaben erkannt, während Variablen immer kleingeschrieben sein müssen. Sonderzeichen wie *Leerzeichen*, *Zeilenende* und *Tabulatoren* werden entfernt und die aktuelle Position im Quelltext wird angepasst.

4.1.3 Parser

Um mit *Happy* einen Parser als Haskellmodul generieren zu können, muss eine Parserspezifikationsdatei definiert werden. In dieser Datei wird zu Beginn der Name des Parsers und der Datentyp der Token definiert:

```
> %name parse
> %tokentype { TokenType }
```

Als nächstes werden alle erlaubten Zeichen und Schlüsselwörter den entsprechenden Token-Kategorien zugeordnet, wobei *TokenUnknown* selbst kein Zeichen zugeordnet bekommt und somit stets zu einem Fehler führt:

```
> %token
> var           { TokenVar  _ _ }
> cons          { TokenCons _ _ }
> '|'           { TokenSymbol   "|" _ }
> '\\'          { TokenSymbol   "\\" _ }
> ';'           { TokenSymbol   ";" _ }
> '='           { TokenSymbol   "=" _ }
> '->'          { TokenSymbol   "->" _ }
> '('           { TokenSymbol  "(" _ }
> '{'           { TokenSymbol  "{" _ }
> ')'           { TokenSymbol  ")" _ }
> '}'           { TokenSymbol  "}" _ }
> 'case'        { TokenKeyword "case" _ }
> 'of'          { TokenKeyword "of" _ }
> 'letrec'      { TokenKeyword "letrec" _ }
> 'in'          { TokenKeyword "in" _ }
> 'seq'         { TokenKeyword "seq" _ }
> 'data'        { TokenKeyword "data" _ }
> 'expr'        { TokenKeyword "expression" _ }
```

Damit der Parser später die Bindungsstärke der Symbole kennt und Ausdrücke ordentlich auswerten kann, benötigt er eine Reihenfolge der Symbole. Hierbei gibt die Reihenfolge vor, welche Symbole am stärksten Binden. Der stärkste Operator wird als erstes Definiert und der schwächste als letztes. Zusätzlich wird angegeben ob es sich bei dem Operator um einen links-

oder rechtsassoziativen Operator handelt, oder um einem nicht assoziativen Operator:

```
> %right '->'
> %right '>>='
> %left 'seq'
> %nonassoc 'in'
> %nonassoc 'of'
> %nonassoc 'case'
> %nonassoc 'letrec'
```

Im Hauptteil der Parserspezifikation wird nun die Grammatik definiert, mit der ein Ausdruck überprüft wird. Ein gültiger Ausdruck besteht hierbei aus Datentypdefinitionen und dem Ausdruck selbst. Datentypdefinitionen sind hierbei optional:

```
> SOURCECODE :: {ParserOutput}
> SOURCECODE :  EXPR              { checkParseTree $ ParseTree [] $1 }
>            |  TDEFS  EXPR       { checkParseTree $ ParseTree $1 $2 }
```

In den geschweiften Klammern steht hierbei die Operation, die auf das Ergebnis angewendet werden soll. Als nächsten werden die Datentypen selbst und Ausdrücke definiert:

```
> TDEFS    : TDEFSIT                 { checkTDEFS $1 }
> TDEFSIT  : TDEF                    { [$1] }
>          | TDEF TDEFSIT            { $1:$2 }
> TDEF     : 'data' cons '=' CDEFS   { (TDef
>                                          (mkCons $2)
>                                          (listsToTypes $4 (TC $ mkCons $2))) }
> CDEFS    : CDEF                    { [$1] }
>          | CDEF '|' CDEFS          { $1:$3 }
> CDEF     : cons                    { (mkCons $1,[]) }
>          | cons TYPE               { (mkCons $1,$2) }
> TYPE     : TK                      { [$1] }
>          | TK TYPE                 { $1:$2 }

> TK       : cons                    { TC $ mkCons $1 }
>          | '(' TK '->' TK ')'      { $2 :->: $4 }
```

```
> EXPR    :: {Expr () ConsName VarName}
> EXPR    : 'expr' REXPR                          { $2 }
> REXPR   : 'letrec' BINDS 'in' REXPR             { checkBinds $4 $2 $1 }
>         | '\\' var '->' REXPR                   { Lam (mkVar $2) $4 }
>         | AEXPR                                 { $1 }

> AEXPR   : IEXPR                                 { $1 }
>         | cons CONSARGS                         { Cons ((mkCons $1)) $2 }
>         | IEXPR CONSARGS                        { mkAPP $1 $2}
>         | cons                                  { Cons ((mkCons $1)) [] }

> BINDS   : BIND                                  { [$1] }
>         | BIND ';' BINDS                        { $1:$3 }

> BIND    : var '=' REXPR                         { (mkVar $1) :=: $3 }

> IEXPR   : 'seq' IEXPR IEXPR                     { Seq $2 $3}
>         | '(' 'seq' cons cons ')'
{ Seq ( Cons ((mkCons $3)) []) ( Cons ((mkCons $4)) [])}
>         | '(' REXPR ')'                         { $2 }
>         | var                                   { Var (mkVar $1) }
>         | 'case' REXPR 'of' '{' ALTS '}'        { checkAlts $1 $2 $5 }
>  ALTS   : ALT                                   { [$1] }
>         | ALT ';' ALTS                          { $1:$3 }
>  ALT    : PAT '->' REXPR
{ checkAlt $2 (CAlt (fst $1) (snd $1) $3) }
>  PAT    : cons                                  { (mkCons $1,[]) }
>         | '(' cons VARARGS ')'                  { (mkCons $2,$3) }
>  VARARGS : var                                  { [mkVar $1] }
>          | var VARARGS                          { (mkVar $1):$2}

>  CONSARGS : IEXPR                               { [$1] }
>           | IEXPR CONSARGS                      { $1:$2}
>           | cons CONSARGS                       { Cons (mkCons $1) []:$2}
>           | cons                                { [Cons (mkCons $1) []]}
```

Neben den Grammatiken müssen Funktionen definiert werden, mit denen Ausdrücke und Datentypdefinitionen auf ihre Richtigkeit überprüft werden können. Dazu gehört eine Funktion die überprüft, ob alle *Case-Alternativen* richtig definiert worden sind, also keine doppelten Alternativen genannt werden. Ähnlich verhält es sich mit den *Letrec-Bindungen*, bei denen keine doppelten Variablennamen vergeben werden dürfen. Bei den Datentypdefinitionen muss darauf geachtet werden, dass keine Datentypen mehrfach genannt werden und das keine Datenkonstruktoren mehrfach definiert werden. Die Funktionen die diese Fehler abfangen sind:

- checkTDEFS
- checkBinds
- checkAlts

Die Hauptfunktion des Parser ist *parseProgram*. Diese verwendet den vorher definierten Lexer um eine Zeichenkette in einen *ParseTree* umzuwandeln.

Ein *ParseTree* besteht hierbei aus den Datentypdefinitionen und dem Ausdruck selbst. Dabei stellt der *ParseTree* den Syntaxbaum dar, der am Ende ausgegeben wird:

```
> parseProgram :: String -> ParserOutput
> parseProgram = parse . lexProgram
> parse :: [TokenType] -> ParserOutput
```

```
> data ParseTree a cname vname = ParseTree [TDef cname cname] (Expr a cname vname)
>  deriving(Eq,Show)
> type ParserOutput = ParseTree () ConsName VarName
> type ConsName = (SourceMark,String)
> type VarName  = (SourceMark,String)
```

Zum Schluss stellt die Funktion *checkParseTree* sicher, dass nur vorher definierte Datenkonstruktoren im Ausdruck verwendet werden. Sollte dies nicht der Fall sein wird eine Fehlermeldung ausgegeben zusammen mit dem Namen des Konstruktors und der Position im Quellcode. Dazu wird der Ausdruck komplett durchlaufen und für jeden Datenkonstruktor wird geprüft, ob dieser in den Datentypdefinitionen definiert wurde:

```
> checkParseTree :: ParserOutput -> ParserOutput
> checkParseTree (ParseTree tdefs e) =
>   let
>    (constructors,consinfo) = toConsInfo tdefs
>    go (Var v) = (Var v)
>    go (Lam v e) = Lam v (go e)
>    go (App e1 e2) = App (go e1) (go e2)
>    go (Seq e1 e2) = Seq (go e1) (go e2)
>    go (Cons (cons) args)
>       | (consName cons) `elem` constructors = Cons (cons) (map go args)
>       | otherwise = error $ "Constructor " ++ show (consName cons )
                            ++ " not defined " ++ "\nRow: "
                            ++ show (fst $ consLabel cons) ++ " "
                            ++ "Col: "  ++ show (snd $ consLabel cons)
>    go (Case e alts) = Case (go e) (map goAlt alts)
>    go (Letrec binds e) = Letrec (map (\(x :=: expr) -> (x :=: (go expr)))
                                   binds) (go e)
>    goAlt (CAlt cname vars e)
>     | (consName cname) `elem` constructors = CAlt cname vars (go e)
>     | otherwise = error $ "Constructor " ++ show (consName cname)
                                             ++ " not defined " ++ "\nRow: "
                                             ++ show (fst $ consLabel cname)
                                             ++ " " ++ "Col: "
                                             ++ show (snd $ consLabel cname)
>   in ParseTree tdefs (go e)
> }
```

Mit dieser Parsespezifikationsdatei *Parser.ly* lässt sich mittel Happy nun das Modul *Parser* generieren, dass in der Datei *Parser.hs* beschrieben wird.

4.2 Die Kernbibliothek

In diesem Abschnitt werden die Kernfunktionen für die Normalordnungsreduktion und Analyse von LR-Ausdrücken implementiert. Dieser Kern wird in dem Modul *ImprovementCore* definiert, der in der Datei *ImprovementCore.hs* beschrieben wird. Dieser besteht aus folgenden einzelnen Funktionen die nachfolgend erläutert werden:

- **reduce/reduce'**: Diese Funktion führt einen Normalordnungsreduktionsschritt aus und implementiert sowohl die einzelnen Normalordnungsreduktionen aus 2.12, als auch den *Labeling-Algorithmus* aus 2.13.
- **rename**: Diese Funktion führt eine Umbenennung der Variablen eines LR-Ausdrucks mit Hilfe von neuen Variablennamen durch.
- **getRLN**: Führt eine Normalordnungsreduktion durch und berechnet dabei den RLN-Wert[2]
- **getRLNDetailed**: Führt eine Normalordnungsreduktion durch und berechnet den RLN-Wert detailliert, dass heißt es wird die Anzahl an *beta-*, *case-* und *seq*-Reduktionen getrennt angezeigt.
- **getRLNALL**: Führt eine Normalordnungsreduktion durch und berechnet den RLNALL-Wert
- **getNormalform**: Führt eine vollständige Normalordnungsreduktion durch bis eine WHNF erreicht wird oder ein Fehler die Reduktion beendet.
- **parseLRFile**: Liest ein LR-Programm aus einer Textdatei mittels Parser und gibt den enthaltenen Ausdruck als LR-Datentyp zurück.
- **createImprovementReport**: Erstellt einen ausführlichen Bericht über das Reduktionsverhalten eines LR-Ausdrucks und gibt diesen als Textdatei aus.
- **analyseRLNofPT**: Berechnet den RLN-Wert eines LR-Ausdrucks vor und nach der Anwendung einer separaten Programmtransformation auf den Ausdruck und gibt beide Werte als Konsolenausgabe aus.
- **createImprovementComparisonReport**: Parst zwei LR-Programme, die jeweils in einer Textdatei beschrieben wurden, und erstellt einen Bericht über die einzelnen RLN-Werte der Ausdrücke.

[2]Siehe 2.3

4.2.1 Die Normalordnungsreduktion

Bevor die eigentliche Normalordnungsreduktion implementiert werden kann, muss zunächst ein Datenkonstruktor implementiert werden, der den aktuellen Stand der Reduktion angeben kann. Es muss bei jedem Reduktionsschritt klar sein, welche Reduktionsregel gerade angewendet wurde oder ob es sich bereits um die fertige WHNF handelt:

```
data ReductionRuleType = Lbeta
| Lapp
| Lseq
| SeqC
| SeqInE
| Lcase
| CaseC
| CaseInE
| CP
| Llet
| WHNF -- ^ Indicates that the reduction has terminated in a WHNF
 deriving(Eq,Show)
```

Die Funktion *reduce* führt einen Normalordnungsreduktionsschritt aus. Die Eingabe der Funktion ist eine Liste mit *neuen Variablennamen*, für den Fall dass ein Ausdruck umbenannt werden muss, und der *LR-Ausdruck* der reduziert werden soll. Als Rückgabe liefert die Funktion ein Tupel der Form:

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

Dieses Tupel besteht aus dem erfolgreich reduzierten Ausdruck, falls der Reduktionsschritt erfolgreich durchgeführt worden ist, zusammen mit der verwendeten Reduktionsregel und den restlichen unbenutzten Variablennamen. Dabei wird hierfür der Datentyp *Maybe* verwendet, um die beiden Fälle der erfolgreichen und nicht erfolgreichen Auswertung unterscheiden zu können. Bei einer nicht erfolgreichen Auswertung wird

(**Nothing**,*Liste mit unbenutzten Variablennamen*)

zurückgegeben, sonst

(**Just (*Ausdruck*,*verwendete Reduktionsregel oder WHNF*)**,*Liste mit unbenutzten Variablennamen*)

Mit Hilfe dieser Konstrukte können nun die ersten Fälle abgearbeitet werden:

```
reduce freshVars (Var x) = (Nothing, freshVars)

reduce freshVars (Lam x e) = (Just (Lam x e,WHNF), freshVars)

reduce freshVars (Cons cname args) = (Just (Cons cname args,WHNF), freshVars)
```

Der einfachste Fall ist ein Ausdruck, der nur aus einer Variablen besteht. An dieser Stelle kann die Reduktion sofort abgebrochen werden, da dieser nicht zu einer WHNF reduziert werden kann.
Die nächsten beiden Fälle sind Ausdrücke die bereits in Form einer WHNF sind und somit bereits fertig reduziert sind. An dieser Stelle kann der Ausdruck selbst zurückgegeben werden zusammen mit der Markierung WHNF.

Die nächsten Fälle sind für *Applikationen*. Hier müssen die Ausdrücke gemäß der Normalordnungsreduktionsregeln *lapp* und *lbeta* bearbeitet werden:

```
--lapp
reduce freshVars (App (Letrec env t) s) =
                        (Just (Letrec env (App t s),Lapp),freshVars)

--lbeta
reduce freshVars (App (Lam x s) r) =
                        (Just (Letrec [x :=: r] s, Lbeta), freshVars)


reduce freshVars (App e1 e2) =
case reduce freshVars e1 of
    (Nothing, freshVars')                          -> (Nothing, freshVars')
    (Just (_,WHNF), freshVars')                    -> (Nothing, freshVars')
    (Just (e1',reductionruletype), freshVars') ->
                        (Just (App e1' e2, reductionruletype), freshVars')
```

Sollte keine der beiden Regeln angewendet werden können, so muss gemäß dem *Labeling-Algorithmus*[3] die Reduktion mit dem ersten Ausdruck der Applikation fortgesetzt werden.

[3] Siehe 2.13

Wenn es sich bei dem Ausdruck um ein *Letrec* handelt, so muss zuerst geprüft werden ob es sich um zwei verschachtelte *Letrec* handelt, die mit der Reduktionsregel *llet-in* reduziert werden können:

```
--llet-in
reduce freshVars (Letrec env1 (Letrec env2 r)) =
                (Just (Letrec (env1 ++ env2) r,Llet),freshVars)
```

Ansonsten muss zunächst der *IN*-Ausdruck des *Letrec* reduziert werden. Sollte dieser zu einer WHNF reduzieren, kann hier die Reduktion beendet werden:

```
reduce freshVars (Letrec env1 e) =
        case reduce freshVars e of
        (Just (e',WHNF),freshVars') -> (Just (Letrec env1 e',WHNF),freshVars')
```

Sollte sich der Ausdruck nicht reduzieren lassen, beziehungsweise sollte die Reduktion des *IN*-Ausdruck mit einer freien Variablen enden, muss diese mit Hilfe einer weiteren Funktion im Ausdruck gesucht werden. Diese Aufgabe wird von der Funktion *findNOVar* erledigt. Diese Funktion bekommt einen Ausdruck als Eingabe und gibt folgende Informationen der ersten gefundenen freien Variablen als Tupel zurück:

- Der Variablenname selbst
- Der gesamte Kontext in dem die Variable auftaucht (als Funktion)
- Der Kontext in dem die Variable auftaucht, aber eine Ebene drüber (als Funktion).
- Der Ausdruck in dem die Variable direkt auftaucht

Ein Beispielausgabe für die *findeNOVar* Funktion sieht wie folgt aus:

Eingabeausdruck:
Seq (Seq (Var "x") (App (Var "y") (Var "z"))) (Var "w")

Die erste freie Variable ist "x".

Ausgabe als Tupel:

- "x"
- Die Funktion: \X -> Seq (Seq X (App (Var "y") (Var "z"))) (Var "w")
- Die Funktion: \X -> Seq X (Var "w")
- Der Ausdruck: Seq (Var "x") (App (Var "y") (Var "z"))

Die Funktionen die Kontexte repräsentieren können hierbei als Ausdrücke mit *Loch* verstanden werden, ähnlich wie bei den Kontexten aus 2.8. Diese Informationen sind notwendig, um mit der nächsten Funktion den *Letrec*-Ausdruck selbst auswerten zu können. Da eine freie Variable entdeckt wurde, muss der Ausdruck für den die freie Variable steht in den Bindungen des *Letrec* gesucht werden. Diese Aufgabe und die Anwendung weiterer Reduktionsregeln, die sich auf *Letrec*-Ausdrücke beziehen, wird von der Funktion *reduce'* durchgeführt.

Die *reduce'* Funktion verwendet die Ausgaben der *findNOVar* Funktion, zusammen mit den *Bindungen* des *Letrec*-Ausdrucks und einer Liste in der schon besuchte freie Variablen gespeichert werden, um den passenden Ausdruck zu der gefundenen freien Variable einzusetzen. Hierbei müssen folgende Fälle beachtet werden. Aus Platzgründen ist die Funktion nicht in dieser Arbeit abgedruckt. Diese kann im Quellcode nachgeschlagen werden:

Zuerst wird in den Bindungen des *Letrec* nach der gefundenen freien Variable und dessen Ausdruck gesucht. Der Ausdruck der Variable wird mit der *exprOfBinding* Funktion extrahiert und verarbeitet.

Wenn es sich bei dem Ausdruck um eine Abstraktion handelt, also ein Lambda, können sofort die *CP*-Reduktionsregeln angewendet werden, in dem die Abstraktion umbenannt wird und zusammen mit dem gesamten Kontext zurückgegeben wird.

Handelt es sich bei dem Ausdruck um eine weitere Variable, so muss nach dem Ausdruck der neuen Variable in den Bindungen gesucht werden. An dieser Stelle kommt die Liste ins Spiel, in der schon besuchte Variablen registriert werden. Wenn die Variable schon besucht worden ist, dann wird eine Fehlermeldung ausgegeben, in der auf unerlaubte zyklische

Variablenbindungen aufmerksam gemacht wird. Dieser Fall sorgt dafür, dass Variablenverkettungen $\{x_i = x_{i-1}\}$[4] definiert werden können und unerlaubte zyklische Definitionen einen Fehler verursachen.

Sollte der Ausdruck ein *Letrec* sein, so muss die Reduktionsregel *llet-e* angewendet werden, in der die beiden verschachtelten *Letrec*-Bindungen zusammengefasst werden und ein zusammengesetzter *Letrec*-Ausdruck entsteht.

Wenn es sich bei dem Ausdruck um einen Konstruktor handelt, so müssen die Reduktionsregeln *seq-in*, *seq-e*, *case-in* und *case-e* in Betracht gezogen werden. Zuerst wird geprüft in welchem direkten Kontext die freie Variable gefunden wurde. Wenn der Kontext ein *seq* ist, so können die Regeln *seq-e* und *seq-in* angewendet werden. Sollte es sich um ein *case* handeln, so muss zuerst die Anzahl der Argumente des Konstruktors betrachtet werden. Wenn der Konstruktor keine Argumente hat, so kann direkt der Ausdruck in der passenden *Case*-Alternative mit vorherigem Kontext ausgegeben werden. Sollte der Konstruktor ein oder mehr Argumente haben, so müssen zusätzlich neue Variablenbindungen den *Case*-Alternativen zugeordnet werden. Zum Schluss kann der *Case*-Ausdruck zu einem *Letrec*-Ausdruck umgebaut werden.

Sollte der gefundene Ausdruck selbst nicht zu den bereits genannten Fällen gehören, so muss geprüft werden ob dieser reduziert werden kann. In diesem Fall wird der Ausdruck nach den gleichen Regeln der Reduktion rekursiv bearbeitet.

Sollte die Variable nicht in den Bindungen gefunden worden sein, so wird eine Fehlermeldung ausgegeben und es wird die undefinierte Variable ausgegeben.

Nachdem die Fälle für *Letrec*-Ausdrücke bearbeitet worden sind, müssen noch die restlichen Fälle der *reduce* Funktion abgehandelt werden. Als nächstes werden *Case*-Ausdrücke bearbeitet. Im ersten Fall kann mit der *lcase*-Regel ein *Letrec*-Ausdruck in einem *Case*-Ausdruck aus dem *Case*-Ausdruck herausgezogen werden:

[4] Siehe 2.12

```
--lcase
reduce freshVars (Case (Letrec env t) alts) =
 (Just (Letrec env (Case t alts),Lcase),freshVars)
```

Der nächste Fall implementiert die Regel *case-c*, in der aus einem *Case*-Ausdruck ein *Letrec*-Ausdruck gemacht wird. Diese Regel ist analog zu den *case-in* und *case-e* Regeln mit dem Unterschied, das der Konstruktor bereits im *Case*-Ausdruck ist und somit direkt ausgewertet werden kann.

Wenn der Ausdruck keinen der genannten Fälle erfüllt, dann muss die Eingabe des *Case*-Ausdruck selbst reduziert werden:

```
reduce freshVars (Case e a) =
       let (e1, freshVars') = reduce freshVars e
        in case e1 of
         Nothing
           -> (Nothing, freshVars')
         Just (e2,reductionruletype)
           ->
                  (Just (Case e2 a,reductionruletype), freshVars')
```

Die Fälle sind die Regeln *lseq* und *seq-c*. Die *lseq*-Regel beschreibt den Fall, dass ein *Letrec* im ersten Argument des *Seq* beschrieben wird. Auch hier kann das *Letrec* herausgezogen werden:

```
--lseq
reduce freshVars (Seq (Letrec env s) t) =
                              (Just (Letrec env (Seq s t),Lseq),freshVars)
```

Die *seq-c*-Regel bearbeitet ein *Seq* allgemein, dass heißt wenn das erste Argument ein Konstruktor oder eine Abstraktion ist, dann kann direkt das zweite Argument des *Seq* ausgegeben werden. Sollte das nicht der Fall sein, dann muss das erste Argument des *Seq* reduziert werden:

```
--seq-c
reduce freshVars (Seq v t) =
 case v of Cons cname args -> (Just (t,SeqC),freshVars)
           Lam x e -> (Just (t,SeqC),freshVars)
           v ->
            case (reduce freshVars v) of
             (Just (v',reductionruletype),freshVars')
               -> (Just (Seq v' t,reductionruletype),freshVars')
```

```
        (Nothing, freshVars')
          -> (Nothing, freshVars')
```

Zum Schluss wird noch sichergestellt, dass Ausdrücke die bisher von keinem der Fälle abgefangen worden sind, eine Fehlermeldung verursachen:

```
reduce v x = error ("Could not reduce: " ++ show x)
```

Die vollständige Normalordnungsreduktion kann nun mit Hilfe der Funktion *getNormalform* durchgeführt werden. Da ein einzelner Reduktionsschritt durch die Funktion *reduce* durchgeführt wird, muss diese solange durchgeführt werden bis eine WHNF erreicht wurde. Dabei muss der Ausdruck selbst zuerst umbenannt werden, damit Namenskonflikte vermieden werden:

```
getNormalform expr =
let (renamedExpr, freshVars') =
        rename expr ["nv_" ++ (show i) | i <- [1..]]
 in getNormalform' renamedExpr freshVars'

getNormalform' expr freshVars =
        case reduce freshVars expr of
        (Nothing, freshVars')                          -> expr
        (Just (expr',WHNF), freshVars')                -> expr'
        (Just (expr', reductionruletype), freshVars')
                          -> getNormalform' expr' freshVars'
```

Die Umbenennung selbst wird von der Funktion *rename* durchgeführt. Der Ausdruck wird hierbei komplett durchlaufen und jede gebundene Variable wird durch eine neue Variable ersetzt (umbenannt). Dabei wird jede Umbenennung in einer Liste gespeichert, damit schon umbenannte Variablen einfach gefunden werden können. Die Umbenennung lässt sich in drei Hauptteile eingliedern. Die Umbenennung von gebundenen Variablen einer Abstraktion, die Umbenennung von *Letrec*-Bindungen und die Umbenennung von *Case*-Alternativen.

Bei gebundenen Variablen einer Abstraktion wird einfach die gebundene Variable durch eine neue ersetzt und die Umbenennung wird in einer Liste gespeichert.

```
rename' varMapping (Lam x e) (v:freshVars) =
 let (e', freshVars') = rename' ((x,v):varMapping) e freshVars
  in (Lam v e', freshVars')
```

Letrec-Bindungen selbst werden wie folgt umbenannt. Zuerst werden alle Bindungsvariablen umbenannt und in der Liste gespeichert. Danach wird jeder Ausdruck der Bindungen rekursiv durchlaufen und ebenfalls umbenannt. Hierbei werden Bindungsvariablen die schon im ersten Schritt umbenannt worden sind in der Liste nachgeschlagen und je nach Resultat neu umbenannt oder mit dem gefundenen Variablennamen Umbenannt.

Bei *Case*-Alternativen müssen jeweils die Konstruktorargumente der Alternative umbenannt werden.

Bei dem Rest der Funktion *rename* wird je nach Konstrukt der Ausdruck rekursiv durchlaufen. Sollte eine Variable gefunden werden, so wird diese in der Liste der Umbenennungen gesucht und entweder neu Umbenannt, oder mit der Variable aus der Liste ersetzt:

```
rename' varMapping (Var x) freshVars =
        case lookup x varMapping of
                Nothing -> (Var x,freshVars)
                Just y  -> (Var y,freshVars)
```

4.2.2 Analyse und Visualisierung

Bei der Berechnung des *RLN*-, *RLNALL*- und des detaillierten *RLN*-Wertes wird jeweils die Funktion *reduce* verwendet, um mit Hilfe des definierten *ReductionRuleType*, die verwendeten Reduktionsregeln zu ermitteln und dadurch den gewünschten Wert zu berechnen.

Der *RLN*-Wert wird mit Hilfe der Funktion *getRLN* berechnet. Diese führt die Normalordnungsreduktion Schritt für Schritt durch und zählt den *RLN*-Wert hoch, wenn eine der folgenden Reduktionsregeln verwendet wurde:

```
getRLN expr = getRLN' expr ["y_" ++ (show i) | i <- [1..]] 0

getRLN' expr freshVars rln =
case (reduce freshVars expr) of
    (Nothing,freshVars')                              -> rln
    (Just (_,WHNF),freshVars')                        -> rln
    (Just (expr',Lbeta),freshVars')
                              -> getRLN' expr' freshVars' (rln+1)
    (Just (expr',Lseq),freshVars')
                              -> getRLN' expr' freshVars' (rln+1)
    (Just (expr',Lcase),freshVars')
```

```
                              -> getRLN' expr' freshVars' (rln+1)
    (Just (expr',SeqC),freshVars')
                              -> getRLN' expr' freshVars' (rln+1)
    (Just (expr',SeqInE),freshVars')
                              -> getRLN' expr' freshVars' (rln+1)
    (Just (expr',CaseC),freshVars')
                              -> getRLN' expr' freshVars' (rln+1)
    (Just (expr',CaseInE),freshVars')
                              -> getRLN' expr' freshVars' (rln+1)
    (Just (expr',reductionruletype),freshVars')
                              -> getRLN' expr' freshVars' rln
```

Der *RLNALL*-Wert wird mit der Funktion *getRLNALL* berechnet. Hier werden alle Reduktionsregeln gezählt:

```
getRLNALL expr = getRLNALL' expr ["y_" ++ (show i) | i <- [1..]] 0

getRLNALL' expr freshVars count =
case (reduce freshVars expr) of
  (Nothing,freshVars') -> count
  (Just (_,WHNF),freshVars') -> count
  (Just (expr',reductuionruletype),freshVars')
      -> getRLNALL' expr' freshVars' (count + 1)
```

Beim detaillierten *RLN*-Wert werden, wie beim *RLN*-Wert, nur bestimmte Reduktionsregeln gezählt. An dieser Stellen werden die Reduktionsregeln nochmal separat voneinander angegeben, um einen genauen Überblick über die Anzahl an *lbeta*-, *case*- und *seq*-Regeln zu bekommen. Die Ausgabe erfolgt als *Triple* statt als einzelne Zahl:

```
getRLNDetailed expr = getRLNDetailed' expr ["y_" ++ (show i) | i <- [1..]] (0,0,0)

getRLNDetailed' expr freshVars (b,c,s) =
case (reduce freshVars expr) of
(Nothing,freshVars')                        -> (b,c,s)
(Just (_,WHNF),freshVars')                  -> (b,c,s)
(Just (expr',Lbeta),freshVars')
                        -> getRLNDetailed' expr' freshVars' (b+1,c,s)
(Just (expr',Lseq),freshVars')
                        -> getRLNDetailed' expr' freshVars' (b,c,s+1)
(Just (expr',Lcase),freshVars')
                        -> getRLNDetailed' expr' freshVars' (b,c+1,s)
(Just (expr',SeqC),freshVars')
                        -> getRLNDetailed' expr' freshVars' (b,c,s+1)
(Just (expr',SeqInE),freshVars')
                        -> getRLNDetailed' expr' freshVars' (b,c,s+1)
(Just (expr',CaseC),freshVars')          -
                        -> getRLNDetailed' expr' freshVars' (b,c+1,s)
(Just (expr',CaseInE),freshVars')
                        -> getRLNDetailed' expr' freshVars' (b,c+1,s)
(Just (expr',reductionruletype),freshVars')
                        -> getRLNDetailed' expr' freshVars' (b,c,s)
```

Die erste Funktion die sich um eine verbesserte Ausgabe kümmert ist die Funktion *showExpression*. Hier wird ein Ausdruck vom Typ *Expr label cname v* verarbeitet und in einer für den Nutzer freundlichere Schreibweise dargestellt:

```
showExpression :: (Eq label, Eq cname,Eq v, Show v, Show label, Show cname)
       => Expr label cname v -> [Char]
showExpression (Var v) = (show v)
showExpression (Lam v e) = "(\\ " ++ (show v) ++ " -> " ++
showExpression e ++ ")"
showExpression (App e1 e2) = "(" ++ showExpression e1 ++ " "
                ++ showExpression e2 ++ ")"
showExpression (Letrec binds e) =
  "(letrec " ++ showBinds ++ " in " ++ showExpression e ++ ")"
  where showBinds = concat . intersperse "; " . map (\(x:=:e) -> (show x)
                ++ " = " ++ showExpression e) $ binds
showExpression (Seq e1 e2) = "(seq " ++ showExpression e1 ++ " "
                ++ showExpression e2 ++ ")"
showExpression (Cons name []) = show name
showExpression (Cons name args) = "(" ++ (show name) ++ " " ++ showArgs ++ ")"
  where showArgs =  concat . intersperse " " . map showExpression $ args
showExpression (Case e alts) = "(case " ++ showExpression e ++ " of "
                ++ "{" ++  showAlts ++ "})"
  where showAlts =  concat . intersperse ";" . map (\(CAlt cname vars expr) ->
       (if null vars then (show cname) else "("++ (show cname) ++ " "
        ++ (concat $ intersperse " " (map show vars)) ++ ")") ++ " -> "
         ++ showExpression expr) $ alts
showExpression (Label l e) = "LABEL" ++ show l ++ "(" ++ showExpression e ++ ")"
```

So wird beispielsweise

```
Lam "x"(Var "x"))
```

als

```
\\x -> x
```

dargestellt.

Die Berichte der Analysen in Textformat werden mit den folgenden Funktionen generiert. Die Funktion *createImprovementReport* generiert einen Bericht mit detaillierten Informationen zur Analyse. Hierbei geht es um einen einzigen Ausdruck der wie folgt verarbeitet wird:

- Der Ausdruck wird umbenannt
- Es werden *RLN*, *RLNALL* und *RLNDetailled* berechnet. Die Werte werden einzeln visualisiert.

- Der Ausdruck wird reduziert. Jeder einzelne Reduktionsschritt und die verwendete Reduktionsregel wird erfasst und abgedruckt
- Alle Informationen werden in einer Textdatei visualisiert

Die Funktion sieht wie folgt aus und besteht aus zwei Teilen. Das Stichwort 'LINE' beschreibt hierbei eine Zeichenkette der Form '========....' die zur Abtrennung der Bereiche dient und in diesem Auszug weggelassen wurde:

```
createImprovementReport' :: Expr () [Char] [Char] -> Handle -> IO ()
createImprovementReport' expr hdl =
        let (renamedExpr,freshVars') = rename expr ["nv_" ++ (show i) | i <- [1..]]
        rln                          = getRLN expr
        rlnall                       = getRLNALL expr
        (b,c,s)                      = getRLNDetailed expr
    in do hPutStrLn hdl $ LINE
          hPutStrLn hdl $ "\nReducing Expression:"
          hPutStrLn hdl $ ("\n\n" ++ (show expr) ++ "\n\n")
          hPutStrLn hdl $ "\nRenamed to:"
          hPutStrLn hdl $ ("\n\n" ++ (show renamedExpr) ++ "\n\n")
          hPutStrLn hdl $ LINE
          hPutStrLn hdl $ "\nImprovement Analysis:"
          hPutStrLn hdl $ ("\n\nRLN: "++(show rln))
          hPutStrLn hdl $ ("\nRLNALL: "++(show rlnall))
          hPutStrLn hdl $ ("\nRLNDetailed (Beta,Case,Seq): "++(show (b,c,s)))
          hPutStrLn hdl $ LINE
          hPutStrLn hdl $ "\nStep-by-Step reduction:"
         createImprovementReport'' renamedExpr freshVars' hdl 1


createImprovementReport'' expr freshVars hdl counter  =
        case reduce freshVars expr of
         (Nothing, freshVars') ->
                    do hPutStrLn hdl $ LINE
                       hPutStrLn hdl $ "\nExpression could not be reduced to WHNF:"
                       hPutStrLn hdl $ (show expr)
                       hPutStrLn hdl $ LINE
         (Just (expr',WHNF),freshVars) ->
                    do hPutStrLn hdl $ LINE
                       hPutStrLn hdl $ "\nReduction ended with WHNF:"
                       hPutStrLn hdl $ (show expr')
                       hPutStrLn hdl $ LINE
         (Just (expr',reductionruletype), freshVars') ->
                    do hPutStrLn hdl $ LINE
                       hPutStrLn hdl $ ("\n"++ show counter ++ ". Step")
                       hPutStrLn hdl $ ("\nReduced to: ")
                       hPutStrLn hdl $ (show expr')
                       hPutStrLn hdl $ ("\nWith Rule: "++ show reductionruletype)
                       hPutStrLn hdl $ LINE
                       createImprovementReport'' expr' freshVars' hdl (counter+1)
```

Im zweiten Teil erkennt man, wie der Ausdruk Schritt für Schritt reduzuert wird und jeder einzelne Schritt visualisiert wird. Je nach Ausgang der

Reduktion wird dann das Resultat ausgegeben. Ein Bericht der mit dieser Funktion erzeugt wurde sieht dann wie folgt aus:

Eingabeausdruck: **(Letrec ["x":=: Lam "y"(Var "y")] (Var "x"))**

Ausgabe:

```
=============================================================

Reducing Expression:

Letrec ["x" :=: Lam "y" (Var "y")] (Var "x")


Renamed to:

Letrec ["nv_1" :=: Lam "nv_2" (Var "nv_2")] (Var "nv_1")


=============================================================

Improvement Analysis:

RLN: 0

RLNALL: 1

RLNDetailed (Beta,Case,Seq): (0,0,0)

=============================================================

Step-by-Step reduction:

=============================================================

1. Step

Reduced to:
Letrec ["nv_1" :=: Lam "nv_2" (Var "nv_2")] (Lam "nv_3" (Var "nv_3"))

With Rule: CP

=============================================================

=============================================================

Reduction ended with WHNF:
Letrec ["nv_1" :=: Lam "nv_2" (Var "nv_2")] (Lam "nv_3" (Var "nv_3"))

=============================================================
```

Man beachte das *CP* erneut eine Umbenennung durchführt.
Die zweite Funktion die einen Bericht erstellt ist *createImprovementComparisonReport*. Diese Vergleicht zwei LR-Programme die als Textdateien vorliegen und erstellt einen Bericht, in dem das Reduktionsverhalten gegenübergestellt wird. Dieser Bericht wird mit folgenden Schritten erzeugt:

- Beide Ausdrücke aus den Textdateien lesen mit Hilfe des Parser
- Beide Ausdrücke umbenennen
- Beide Ausdrücke reduzieren und in Normalform zeigen
- Für beide Ausdrücke *RLN*, *RLNALL* und *RLNDetailled* berechnen und anzeigen

```
createImprovementComparisonReport :: [Char] -> [Char] -> IO ()
createImprovementComparisonReport lrfile1 lrfile2 =
    do program1 <- readFile lrfile1
       program2 <- readFile lrfile2
       hdl <- openFile "ImprovementComparisonReport.txt" (WriteMode)
       let expr1 = parseExpression program1
           expr2 = parseExpression program2
           expr1Renamed = fst (rename expr1 ["nv_" ++ (show i) | i <- [1..]])
           expr2Renamed = fst (rename expr2 ["nv_" ++ (show i) | i <- [1..]])
           rln1 = getRLN expr1Renamed
           rln2 = getRLN expr2Renamed
           rlnall1 = getRLNALL expr1Renamed
           rlnall2 = getRLNALL expr2Renamed
           rlndetailled1 = getRLNDetailed expr1Renamed
           rlndetailled2 = getRLNDetailed expr2Renamed
           nf1 = getNormalform expr1
           nf2 = getNormalform expr2
       in do hPutStrLn hdl $ LINE
             hPutStrLn hdl $ ("\nComparison␣Results␣of␣"++(show lrfile1)
                       ++"␣and␣"++(show lrfile2)++":")
             hPutStrLn hdl $ LINE
             hPutStrLn hdl $ ("\n"++(show lrfile1)++":")
             hPutStrLn hdl $ ("\nExpr:␣")
             hPutStrLn hdl $ ("\n"++(show expr1))
             hPutStrLn hdl $ ("\nExpr␣Renamed:␣")
             hPutStrLn hdl $ ("\n"++(show expr1Renamed))
             hPutStrLn hdl $ ("\nNormalForm:␣")
             hPutStrLn hdl $ ("\n"++(show nf1))
             hPutStrLn hdl $ ("\nRLN:␣")
             hPutStrLn hdl $ ("\n"++(show rln1))
             hPutStrLn hdl $ ("\nRLNALL:␣")
             hPutStrLn hdl $ ("\n"++(show rlnall1))
             hPutStrLn hdl $ ("\nRLNDetailed␣(Beta,Case,Seq):␣")
             hPutStrLn hdl $ ("\n"++(show rlndetailled1))
             hPutStrLn hdl $ LINE
             hPutStrLn hdl $ ("\n"++(show lrfile2)++":")
             hPutStrLn hdl $ ("\nExpr:␣")
             hPutStrLn hdl $ ("\n"++(show expr2))
```

```
hPutStrLn hdl $ ("\nExpr Renamed: ")
hPutStrLn hdl $ ("\n"++(show expr2Renamed))
hPutStrLn hdl $ ("\nNormalForm: ")
hPutStrLn hdl $ ("\n"++(show nf2))
hPutStrLn hdl $ ("\nRLN: ")
hPutStrLn hdl $ ("\n"++(show rln2))
hPutStrLn hdl $ ("\nRLNALL: ")
hPutStrLn hdl $ ("\n"++(show rlnall2))
hPutStrLn hdl $ ("\nRLNDetailed (Beta,Case,Seq): ")
hPutStrLn hdl $ ("\n"++(show rlndetailled2))
hFlush hdl
hClose hdl
```

Der Bericht für die folgenden zwei Ausdrücke sieht dann wie folgt aus:

Erstes LR-Programm (ID_Test.lr):

expression
\x -> x

Zweites LR-Programm (Let_Test.lr):

expression
letrec x = \x -> x;
y = \y -> y
in (x y)

Ausgabe:

```
Comparison Results of "ID_Test.lr" and "Let_Test.lr":

"ID_Test.lr":

Expr:

Lam "x" (Var "x")

Expr Renamed:

Lam "nv_1" (Var "nv_1")
```

```
NormalForm:

Lam "nv_1" (Var "nv_1")

RLN:

0

RLNALL:

0

RLNDetailed (Beta,Case,Seq):

(0,0,0)
```

```
"Let_Test.lr":

Expr:

Letrec ["x" :=: Lam "x" (Var "x"),"y" :=: Lam "y" (Var "y")] (App (Var "x") (Var "y"))

Expr Renamed:

Letrec ["nv_1" :=: Lam "nv_3" (Var "nv_3"),"nv_2" :=: Lam "nv_4" (Var "nv_4")]
                                  (App (Var "nv_1") (Var "nv_2"))

NormalForm:

Letrec ["nv_1" :=: Lam "nv_3" (Var "nv_3"),"nv_2" :=: Lam "nv_4" (Var "nv_4"),
                "nv_5" :=: Var "nv_2"]
                (Lam "nv_6" (Var "nv_6"))

RLN:

1

RLNALL:

4

RLNDetailed (Beta,Case,Seq):

(1,0,0)
```

Mit der Funktion *analyseRLNofPT* kann der Nutzer den *RLN*-Wert seiner eigenen Programmtransformationen schnell berechnen. Dieser wird dann als Zeichenkette angezeigt. Zuerst wird die Programmtransformation angegeben, die in idealerweise im Modul *ProgramTransformations* definiert wurde. Zusätzlich muss ein Ausdruck angegeben werden, der reduziert wird. Schließlich werden die *RLN*-Werte wie folgt berechnet:

- Berechne den *RLN*-Wert nachdem die angegebene Programmtransformation durchgeführt wurde
- Berechne den *RLN*-Wert ohne die angegeben Programmtransformation
- Gebe die beiden Werte aus

```
analyseRLNofPT :: (Expr () [Char] [Char] -> Expr () [Char] [Char])
                                  -> Expr () [Char] [Char] -> IO()
analyseRLNofPT pt expr =
        do let  rln1 =
               (getRLN (pt (fst (rename expr ["nv_" ++ (show i) | i <- [1..]])))))
                rln2 =
               (getRLN (fst (rename expr ["nv_" ++ (show i) | i <- [1..]])))
           in putStrLn ("RLN after transformation: "++(show rln1)
                  ++" and RLN before transformation: "++(show rln2))
```

4.3 Graphische Benutzerschnittstelle

Für die Implementierung einer graphischen Benutzerschnittstelle konnte ein großer Teil aus [Cas13] übernommen werden. Dabei mussten nur noch einzelne Stellen angepasst und ergänzt werden. Der folgende Abschnitt beschränkt sich dabei nur auf die Erläuterung dieser Anpassungen.

Zu Beginn musste das Grundgerüst der Oberfläche, dass mittels Glade[5] erstellt wurde, angepasst werden.

Mit Glade können graphische Benutzeroberflächen entwickelt und in einem XML-Format abgespeichert werden. Diese Oberflächen können dann mittels GTK+[6] gesteuert und genutzt werden. Bei GTK+ handelt es sich um ein von mehreren Plattformen unterstütztes Werkzeug zum Erstellen und Steuern von graphischen Benutzeroberflächen. GTK+ unterstützt eine große Bandbreite an Programmiersprachen, unter Anderem auch Haskell. Mit Hilfe von Haskellmodulen, wie dem *Graphics.UI.Gtk.Builder*[7] Modul, können diese Oberflächen dann mit funktionalen Programmen gesteuert und implementiert werden.[Cas13, S. 37]

Die Oberfläche musste eine zusätzliche Anzeige für die schnelle Ausgabe des *RLN*- und *RLNALL*-Wertes bereitstellen und die neuen Funktionen

[5] Siehe `https://glade.gnome.org/`

[6] Siehe `http://www.gtk.org/`

[7] Siehe `https://hackage.haskell.org/package/gtk-0.14.2/docs/Graphics-UI-Gtk-Builder.html`

mussten im Menu eingebettet werden:

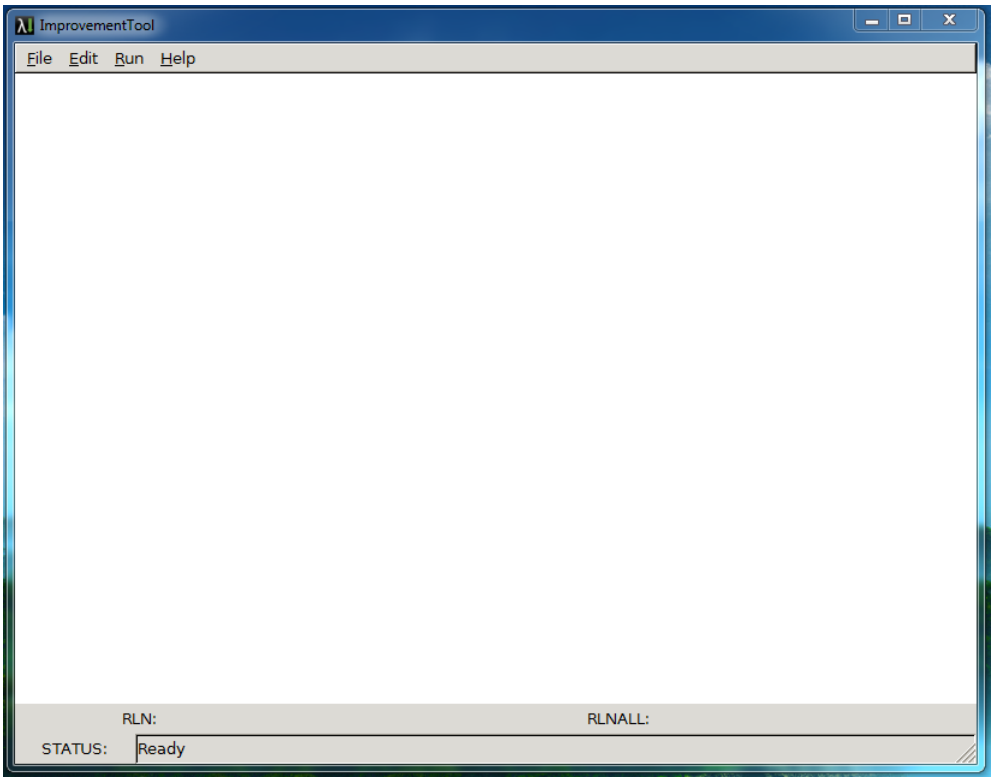

Abbildung 4.1: Angepasste graphische Oberfläche

Die graphische Oberfläche unterstützt folgende Funktionen:

- Die volle Funktionalität eines Texteditors (Lesen, Ändern und Speichern von Dateien)
- Die Berechnung und Anzeige von *RLN* und *RLNALL* eines *LR*-Programms
- Das Erzeugen eines einfachen Berichts über das Reduktionsverhalten eines *LR*-Programms

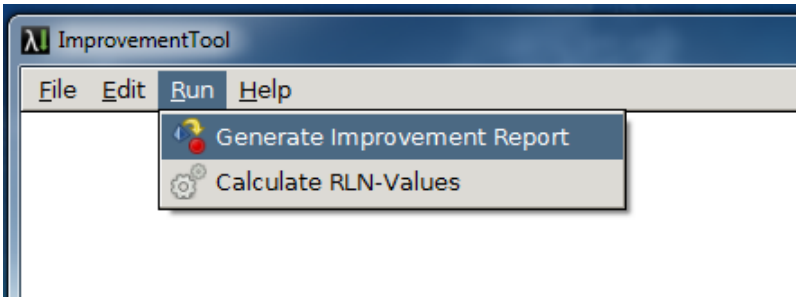

Abbildung 4.2: Die Funktionen der graphischen Oberfläche

Die ursprüngliche Implementierung konnte, an den Stellen an denen die Funktionen aufgerufen wurden, entsprechend angepasst werden:

Laden der XML-Oberfläche mittels *Gtk.Builder*:

```
main = do

    initGUI

    builder <- builderNew
    builderAddFromFile builder "./config/glade/gui.glade"

    mainWindow <- builderGetObject builder castToWindow "window1"
```

Anpassen der Menupunkte und Aufrufen der neuen Funktionen:

```
onActivateLeaf runReportButton $ do
    putStrLn "Analyse LR-Program"
    CE.catch (do
       textbuffer <- textViewGetBuffer textview
       startIt    <- textBufferGetStartIter textbuffer
       endIt      <- textBufferGetEndIter textbuffer
       text       <- textBufferGetText textbuffer startIt endIt True
       let expr   = parseExpression text
        in createImprovementReport expr)
     (\(CE.SomeException err)
       -> (do  setStatusbar statusbar 1
                  ("Could not create Analysis Report. Caused by: "
                ++(show err))
            putStrLn "Error"))


onActivateLeaf runRLNButton $ do
    putStrLn "Calculate RLN Values"
```

```
CE.catch (do
    textbuffer <- textViewGetBuffer textview
    startIt    <- textBufferGetStartIter textbuffer
    endIt      <- textBufferGetEndIter textbuffer
    text       <- textBufferGetText textbuffer startIt endIt True
    let expr   = parseExpression text
        rln    = getRLN expr
        rlnall = getRLNALL expr
     in setRLNValues (show rln) (show rlnall) rlnlabel rlnalllabel)
(\(CE.SomeException err) ->
    (do  setStatusbar statusbar 1
                 ("Could not calculate RLN. Caused by: "
                 ++(show err))
         putStrLn "Error"))
```

Dabei musste jeder Aufruf des XML-Objekts mit Hilfe des *Gtk.Builder* Moduls angepasst werden. Der Rest der Implementierung konnte so übernommen werden.

Nachdem der Kern des Werkzeugs definiert worden ist, kann nun anhand von Tests sichergestellt werden, dass die einzelnen Funktionen und Bestandteile ein korrektes Verhalten zeigen.

4.4 Tests

In diesem Abschnitt werden die Tests des Werkzeugs erläutert. Beim Testen wurden zwei bekannte Testmethoden angewendet, die sich für den jeweiligen Bereich am besten geeignet haben. Die Kernimplementierung wurde mit *Komponententests* überprüft, die in einem eigenen Modul *ImprovementTest* implementiert wurden. Einzelne Teile, wie die Umbenennung oder die Funktionalität der graphischen Oberfläche, wurden in Form von *Akzeptanztests* überprüft.

4.4.1 Das Testmodul

Da sich die Reduktion aus verschiedenen Reduktionsregeln zusammensetzt, konnten einzelne Regeln explizit getestet werden. Für jede implementierte Reduktionsregel wurde eine eigene Testfunktion geschrieben, mit der man das Verhalten der Reduktion überprüfen kann. Da sich viele Reduktionsregeln auf eine unbegrenzte Anzahl an Kontexten beziehen, konnten hierbei nur bestimmte Kontexte getestet werden. Ein Beispiel für eine solche Testfunktion ist die Testfunktion für die *caseC*-Regel:

```
-- ### Test for rule case-c ###
-- c without args
caseCTest = Case (Cons "True" [])
            [CAlt "False" [] (Var "x3"),
```

```
              CAlt "True" [] (Lam "x1" (Var "x1"))]
-- c with args
caseCTest1 = Case (Cons "True" [Var "t1"])
              [CAlt "False" ["y1","y2"] (Var "x3"),
               CAlt "True" ["y1"] (Lam "x1" (Var "x1"))]
-- with context APP
caseCTest2 = App (Case (Cons "True" [Var "t1"])
                    [CAlt "False" ["y1","y2"] (Var "x3"),
                     CAlt "True" ["y1"] (Lam "x1" (Var "x1"))])
                   (Lam "x2" (Var "x2"))
```

Hier mussten die einzelnen Fälle betrachtet werden:

- Konstruktor ohne Argumente
- Konstruktor mit Argument
- Konstruktor mit Argumente und Case-Kontext App

Eingabe:

```
getNormalform caseCTest
```

Erwartete Ausgabe:

```
Lam "nv_2" (Var "nv_2")
```

Tatsächliche Ausgabe:

```
Lam "nv_2" (Var "nv_2") -- ohne Umbenennung: Lam "x1" (Var "x1")
```

Beim definieren der Testfunktionen musste darauf geachtet werden, dass eine ausreichende Abdeckung der Fälle gewährleistet wird. Zugleich musste beachtet werden, dass eine vollständige Abdeckung aller Kontexte nicht möglich ist.

Mit der Funktion *printReductionToFile* wurde während der gesamten Implementierungsphase überprüft, dass die Reduktion korrekt abläuft. Diese Funktion hat die einzelnen Schritte in einer Textdatei visualisiert, die dann überprüft werden konnte. Die Funktion selbst diente ebenfalls als Vorlage für die Implementierung der *createImprovementReport* und

createImprovementComparisonReport Funktionen, mit denen die Berichte visualisiert werden. Zusammen mit der Funktion *testParse* konnte hier der gesamte Ablauf getestet werden. Vom Parsen des LR-Programms bis hin zur Reduktion.

Die Umbenennung wurde nach einem ähnlichen Schema getestet. Mit der Funktion *printRenameToFile* konnte ein Ausdruck vor und nach der Umbenennung in einer Textdatei visualisiert werden, die dann überprüft werden konnte. Zusätzlich wurden zwei Testfunktionen implementiert, die dann getestet werden konnten:

```
renameExpr1 =
Letrec ["x1" :=: Lam "x1" (Var "x1"), "x2" :=: Var "x1"]
                                (App (Seq (Var "x2") (Lam "x1" (Var "x1")))
                                 (Lam "z10" (Var "z10")))
```

Dabei musste immer vorher klargestellt werden, wie eine korrekte Umbenennung des Ausdrucks auszusehen hat:

Eingabe:

```
printRenameToFile renameExpr1
```

Erwartete Ausgabe:

```
Letrec ["nv_1" :=: Lam "nv_3" (Var "nv_3"),"nv_2" :=: Var "nv_1"]
(App (Seq (Var "nv_2") (Lam "nv_4" (Var "nv_4"))) (Lam "nv_5" (Var "nv_5")))
```

Tatsächliche Ausgabe:

```
Expression:

Letrec ["x1" :=: Lam "x1" (Var "x1"),"x2" :=: Var "x1"]
(App (Seq (Var "x2") (Lam "x1" (Var "x1"))) (Lam "z10" (Var "z10")))

Renamed to:

Letrec ["nv_1" :=: Lam "nv_3" (Var "nv_3"),"nv_2" :=: Var "nv_1"]
```

```
(App (Seq (Var "nv_2") (Lam "nv_4" (Var "nv_4"))) (Lam "nv_5" (Var "nv_5")))
```

Spezialfälle, wie zyklische Variablenbindungen oder frei Variablen, wurden mit eigenen Testfällen abgedeckt:

```
cyclicBindsTest = Letrec ["x1" :=: Var "x2", "x2" :=: Var "x1"] (Var "x1")

freeVarTest1 = Letrec ["x1" :=: Var "x3", "x2" :=: Var "x1"] (App (Var "x2")
                                                                 (Lam "x" (Var "x")))
```

Eingabe:

```
getNormalform cyclicBindsTest
```

Erwartete Ausgabe:

Eine Fehlermeldung die besagt, dass wir zyklische Bindungen haben

Tatsächliche Ausgabe:

```
*** Exception: expression has cyclic bindingsVar "nv_1"
-- ohne Umbenennung: "x1"
```

Die letzten Funktionen testen das Verhalten der Programmtransformation *GC*. Diese wird im nächsten Kapitel erläutert.

Kapitel 5

Anwendung des Werkzeugs

In diesem Kapitel werden zwei Anwendungsbeispiele vorgestellt, mit denen die Bedienung des Werkzeugs verdeutlicht werden soll. Zu Beginn wird im Modul *ProgramTransformations* eine neue zusätzliche Programmtransformation implementiert, die mit Hilfe des Werkzeugs analysiert werden kann. Anschließend wird die graphische Oberfläche benutzt, um das Reduktionsverhalten von zwei LR-Programmen zu vergleichen.

5.1 Zusätzliche Programmtransformationen

In [SS15, S. 5] werden eine Reihe zusätzlicher Programmtransformationen vorgestellt. Eine davon ist die *Garbage Collection*:

(gc1) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, \mathit{Env}\ \texttt{in}\ t) \to (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)$ if for all $i : x_i$ does not occur in Env nor in t
(gc2) $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) \to t$ if for all $i : x_i$ does not occur in t

Abbildung 5.1: Die *Garbage Collection* Regel

Mit der *GC*-Regel werden ungenutzte Variablenbindungen eines *Letrec* entfernt. Dabei besteht die Regel aus zwei Teilen. Im ersten Teil werden ungenutzte Variablenbindungen, die nicht in der restlichen Umgebung *Env* und im *IN*-Ausdruck vorkommen, entfernt. Im zweiten Teil wird das komplette *Letrec*-Konstrukt entfernt, falls die gesamte Umgebung nicht im *IN*-Ausdruck vorkommt.

Beide Teile der *GC*-Regel werden durch folgende Funktion implementiert:

```
gc expr = case expr of
    Letrec b e   -> case (gc' b [] [] (deleteDuplicates (findVars e))) of
                    --gc2
                     [] -> e
                    --gc1
                    nB -> Letrec nB e
    otherwise    -> error "Program is not a Letrec"

gc' _ nB _  []          = nB
gc' b nB mem (x:xs)     =
    let expr = getExprVar x b
     in case expr of
          Just (Var v) -> if (elem v mem)
                          then error ("cycle in let-bindings: "++show b)
                          else gc' b (nB++[(x :=: Var v)])
                                     (mem++[x]) (v:xs)
          Just expr    -> gc' b (nB++[(x :=: expr)])
                                (mem++[x])
                                (deleteDuplicates ((findVars expr)++xs))
          Nothing      -> gc' b nB mem xs
```

Die Unterteilung der beiden Teile findet im ersten Funktionsabschnitt statt, dort wird direkt festgestellt, ob tatsächlich Variablen in den Bindungen im *IN*-Ausdruck verwendet werden oder nicht. Im zweiten Teil findet dann die Überprüfung statt. Das Ergebnis der Überprüfung ist eine Liste mit den Variablenbindungen die tatsächlich relevant sind für den *IN*-Ausdruck. Die Eingabe der Funktion *gc'* sind wie folgt:

- Die Bindungen des Letrec *b*
- Eine Liste in der die tatsächlich relevanten Bindungen gespeichert werden *nB*
- Eine Liste die als Merkliste für schon besuchte Variablen dient zur Erkennung von zyklischen Bindungen
- Eine Liste mit den Variablennamen die im *IN*-Ausdruck vorkommen

Zunächst müssen die Variablen die im *IN*-Ausdruck vorkommen extrahiert werden. Diese Aufgabe wird von der *findVars* Funktion durchgeführt:

```
findVars :: Expr () [Char] [Char] -> [[Char]]
findVars (Var v)           = [v]
findVars (Lam x e)         = []
findVars (App e1 e2)       = (findVars e1) ++ (findVars e2)
findVars (Letrec b e)      = []
findVars (Seq e1 e2)       = (findVars e1) ++ (findVars e2)
findVars (Cons cname args) = []
findVars (Case e alts)     = findVars e
```

Hierbei können Variablen die mehrfach vorkommen doppelt genannt werden. Aus diesem Grund wir die Liste direkt mit der Funktion *deleteDuplicates* von Mehrfachnennungen befreit:

```
deleteDuplicates :: [[Char]] -> [[Char]]
deleteDuplicates list        = deleteDuplicates' list []

deleteDuplicates' [] mem     = mem
deleteDuplicates' (x:xs) mem = if elem x mem
                               then deleteDuplicates' xs mem
                               else deleteDuplicates' xs (mem++[x])
```

Nachdem die Variablennamen ausfindig gemacht worden sind, können nun die Bindungen des *Letrec* überprüft werden. Zunächst wird die Variable des *IN*-Ausdruck in den Bindungen gesucht und der Bindungsausdruck wird mit der Funktion *getExprVar* ausgelesen. Sollte es sich erneut um eine Variable handeln, wird geprüft ob diese Variable bereits vorgekommen ist. In diesem Fall handelt es sich um eine zyklische Variablenbelegung und es kann eine Fehlermeldung ausgeben werden. Falls dies nicht der Fall ist, wird die Variable in die Merkliste eingetragen und die komplette Variablenbindung wird zu der Liste der relevanten Variablenbindungen hinzugefügt.
Wenn es sich bei dem Ausdruck um keine Variable handelt, dann wird die gesamte Bindung zu den relevanten Bindungen hinzugefügt, die Variable wird zu der Merkliste hinzugefügt und die Variablen des Bindungsausdrucks (mittels findVars und deleteDuplicates) werden zu der Liste mit den zu untersuchenden Variablen hinzugefügt.

Um die Funktionsweise zu testen wurden erneut Testfunktionen im *ImprovementTest* Modul implementiert. Hierbei müssen folgende Fälle betrachtet werden:

- Es existiert keine relevante Variable (gc2)
- Es existieren sowohl ungenutzte, als auch genutzte Variablen.
- Es existieren sowohl ungenutzte Variablen, als auch Variablen die in der Umgebung selbst genutzt werden
- Es existieren zyklische Variablenbindungen

Keine relevante Variablen::

Eingabe:

```
gc gcExpr
```

Ausdruck:

```
gcExpr = Letrec ["z" :=: (Var "y")] (Var "x")
```

Erwartete Ausgabe:

```
Var "x"
```

Tatsächliche Ausgabe:

```
Var "x"
```

Genutzte Variablen, als auch Ungenutzte Variablen::

Eingabe:

```
gc gcExpr4
```

Ausdruck:

```
gcExpr4 = Letrec ["x" :=: Var "y",
                  "y" :=: Var "z",
                  "unused" :=: Var "z",
                  "unused1" :=: Var "unused"]
                  (App (Var "x")
                       (Seq (App (Var "app1")
                                 (Case (Var "case") []))
                            (Var "seq2")))
```

Erwartete Ausgabe:

```
Letrec ["x" :=: Var "y",
        "y" :=: Var "z"]
        (App (Var "x")
             (Seq (App (Var "app1")
                       (Case (Var "case") []))
                  (Var "seq2")))
```

Tatsächliche Ausgabe:

```
Letrec ["x" :=: Var "y",
        "y" :=: Var "z"]
        (App (Var "x")
             (Seq (App (Var "app1")
                       (Case (Var "case") []))
                  (Var "seq2")))
```

Genutzte Variablen in der Umgebung, als auch ungenutzte Variablen:

Eingabe:

```
gc gcExpr2
```

Ausdruck:

```
gcExpr2 = Letrec ["x" :=: Var "y",
                  "y" :=: Var "z",
                  "z" :=: App (Var "u") (Var "u"),
                  "u" :=: (Var "end"),
                  "unused" :=: Var "z",
                  "unused1" :=: Var "unused"]
(App (Var "x")
     (Seq
         (App (Var "x")
              (Case (Var "case") []))
         (Var "seq2")))
```

Erwartete Ausgabe:

```
Letrec ["x" :=: Var "y",
        "y" :=: Var "z",
        "z" :=: App (Var "u") (Var "u"),
        "u" :=: Var "end"]
        (App (Var "x")
             (Seq (App (Var "x")
                       (Case (Var "case") []))
                  (Var "seq2")))
```

Tatsächliche Ausgabe:

```
Letrec ["x" :=: Var "y",
        "y" :=: Var "z",
        "z" :=: App (Var "u") (Var "u"),
        "u" :=: Var "end"]
        (App (Var "x")
             (Seq (App (Var "x")
                       (Case (Var "case") []))
                  (Var "seq2")))
```

Zyklische Bindungen::

Eingabe:

```
gc gcExpr3
```

Ausdruck:

```
gcExpr3 = Letrec ["x" :=: (Var "y"), "y" :=: (Var "x")] (Var "x")
```

Erwartete Ausgabe:

Eine Fehlermeldung die auf zyklische Bindungen hinweist.

Tatsächliche Ausgabe:

```
*** Exception: cycle in let-bindings: ["x" :=: Var "y","y" :=: Var "x"]
```

5.2 Anwenden der graphischen Oberfläche

Die Funktionsweise der graphischen Oberfläche wird anhand eines Beispiels aus [SSS15, S. 10] demonstriert, in dem folgendes gezeigt wird:

Sei $map = \lambda f,xs.$case xs of (Nil -> Nil)
(y : ys -> (f y) : (map f ys))

Dann ist der Ausdruck:

$s = \mathbf{L}$[map (λx.g (h x)) ls]

eine Optimierung (*Improvement*) von:

$t = \mathbf{L}$[map g (map h ls)]

Dabei ist *map* eine Funktion, die eine weitere Funktion und eine Liste als Eingabe erhält, und die Funktion auf jedes Element der Liste anwendet. Das Resultat dieser Operation ist erneut eine Liste. **L** stellt dabei ein *Letrec* dar, dass alle nötigen Bindungen bereitstellt.

In diesem Abschnitt wird die oben genannte Aussage mit Hilfe des Werkzeugs und dessen graphische Oberfläche überprüft. Dabei mussten folgende Schritte abgearbeitet werden:

- Implementieren der beiden Ausdrücke als eigenständige LR-Programme
- Erstellen der Berichte über das Reduktionsverhalten

5.2.1 Implementierung der LR-Programme

Zu Beginn mussten die beiden Ausdrücke als funktionsfähige LR-Programme in Form von *.LR*-Dateien zur Verfügung gestellt werden. Es mussten alle nötigen Datentypen implementiert werden und der Hauptausdruck. In diesem Fall wurden Listen betrachtet, die Zahlen als Elemente haben. Da Zahlen selbst nicht dargestellt werden können, mussten diese als *Peanozahlen* implementiert werden. Dabei werden Zahlen in Form von Listen repräsentiert. Die Zahl *0* entspricht beispielsweise einer leeren Liste und die Zahl *3* einer Liste mit drei Elementen *('S':('S':('S':[])))*. Die Implementierung der beiden Ausdrücke sieht dann wie folgt aus:

Ausdruck s:

```
data Peano = S Peano | Null
data PeanoList =  Nil | Cons Peano PeanoList
expression
letrec g = \x -> x;
h = \x -> x;
map = \f -> \xs -> case xs of {Nil -> Nil; (Cons y ys) -> Cons (f y) (map f ys)};
p1 = S Null;
p2 = S (S Null);
p3 = S (S (S Null));
list = Cons p1 (Cons p2 (Cons p3 Nil))
in map (\x -> (g (h x))) list
```

Ausdruck t:

```
data Peano = S Peano | Null
data PeanoList =  Nil | Cons Peano PeanoList
expression
letrec g = \x -> x;
h = \x -> x;
map = \f -> \xs -> case xs of
                    {Nil -> Nil; (Cons y ys) -> Cons (f y) (map f ys)};
p1 = S Null;
p2 = S (S Null);
p3 = S (S (S Null));
list = Cons p1 (Cons p2 (Cons p3 Nil))
in map g (map h list)
```

Als Abstraktionen wurde in allen Fällen die *Identitätsfunktion* genommen, um den Ausdruck zu vereinfachen. Die Liste die bearbeitet wurde bestand jeweils aus drei Peanozahlen. Die Liste für Peanozahlen selbst musste ebenfalls als geeigneter Datentyp definiert werden. Die beiden Programme konnten einfach über die Oberfläche erzeugt und abgespeichert werden:

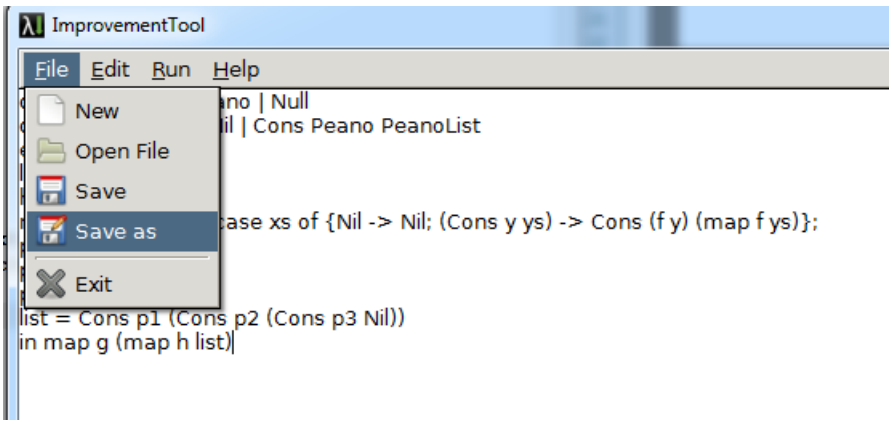

Abbildung 5.2: Erstellen und Speichern von LR-Programmen

5.2.2 Das Reduktionsverhalten

Nachdem die Programme definiert wurden, können die *RLN*-Werte berechnet und die ausführlichen Berichte über das Reduktionsverhalten erzeugt werden. Die *RLN*-Werte konnten direkt über die entsprechende Funktion berechnet werden:

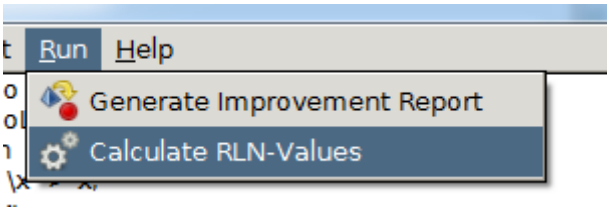

Abbildung 5.3: Berechnen der RLN-Werte

Werte für den Ausdruck *s*:

RLN: 3
RLNALL: 8

Werte für den Ausdruck *t*:

RLN: 6
RLNALL: 16

Die ausführlichen Berichte konnten über die Option *Generate Improvement Report* erzeugt werden. Nachfolgend Auszüge aus den Berichten die zur verbesserten Übersichtlichkeit gekürzt wurden:

Bericht für den Ausdruck *s*:

```
Reducing Expression:
...
Renamed to:
...
```

```
Improvement Analysis:

RLN: 3

RLNALL: 8

RLNDetailed (Beta,Case,Seq): (2,1,0)
```

```
Step-by-Step reduction:
```

```
1. Step

...

With Rule: CP

...
```

Bericht für den Ausdruck t:

```
Reducing Expression:

...

Renamed to:

...
```

```
Improvement Analysis:

RLN: 6

RLNALL: 16

RLNDetailed (Beta,Case,Seq): (4,2,0)
```

```
Step-by-Step reduction:

================================================================================

1. Step

...

With Rule: CP

...
```

Anhand der berechneten *RLN*-Werte erkennt man das der Ausdruck s tatsächlich eine Optimierung des Ausdrucks t ist. Der detaillierte *RLN*-Wert verrät, dass der Ausdruck t doppelt so viele *lbeta*-Reduktionsschritte und *case*-Reduktionschritte benötigt als der s Ausdruck. Die Aussage aus [SSS15, S. 10] konnte mit Hilfe dieses Werkzeugs also bestätigt werden.

Kapitel 6

Zusammenfassung und Fazit

In dieser Arbeit konnte ein Werkzeug entwickelt werden, mit dem man das Reduktionsverhalten von LR-Programmen untersuchen und analysieren kann. Zu Beginn wurden die Voraussetzungen geschaffen, um ein LR-Programm übersetzen zu können. Hierfür wurde mit Hilfe der Grammatik der Kernsprache ein *Parser* implementiert. Um die Auswertung eines solchen Programms zu ermöglichen, wurde eine Normalordnungsreduktion implementiert die Schritt für Schritt einen Ausdruck auswertet und dabei die verwendeten Reduktionsregeln darstellt. Anschließend konnten mit Hilfe der Normalordnungsreduktion überprüft werden, ob es sich bei Programmtransformationen oder Ausdrücken um sogenannte *Improvements* handelt. Hierfür wurde die Anzahl bestimmter Reduktionregeln gezählt und ein ausführlicher Bericht über das Reduktionsverhalten erstellt. Neben der Bereitstellung des Werkzeugs als Bibliothek wurde durch die Implementierung einer graphischen Benutzerschnittstelle eine benutzerfreundliche Alternative zur Verfügung gestellt.

Ein interessantes Thema für zukünftige Arbeiten in diesem Bereich ist der Umgang mit Kontexten. Wie bereits bekannt muss eine Programmoptimierung für jeden Kontext gültig sein. Da es aber eine unbegrenzte Menge an Kontexten gibt müssten in der Theorie alle Kontexte überprüft werden.

Mit diesem Werkzeug ist es gelungen ein weiteres Hilfsmittel bei der Analyse von funktionalen Programmen bereitzustellen. Mit Hilfe der Bibliothek und der graphischen Oberfläche lassen sich Aussagen bezüglich des Reduktionsverhaltens einfach überprüfen. Zusätzlich kann dieses Werkzeug als Grundgerüst dienen, um das Reduktionsverhalten von weiteren funktionalen Kernsprachen zu untersuchen.

Die Forschung im Bereich der Programmiersprachen und dessen Optimierung ist ein interessantes Feld in dem noch viel Potential für die Zukunft steckt. Der Bereich der funktionalen Programmiersprachen bietet hierfür ein hervorragendes Fundament auf dem man neue Ideen und Verbesserungen entwickeln kann, die dann Auswirkungen auf alle Programmiersprachen haben können.

Danksagung

An dieser Stelle möchte ich mich zuerst bei meiner Familie und meinen Freunden bedanken, die mich während meines gesamten Studiums unterstützt und motiviert haben.

Ein besonderer Dank geht an die Professur für Künstliche Intelligenz und Softwaretechnologie der Goethe-Universität Frankfurt, die mich sowohl während meiner Bachelorarbeit, als auch während dieser Masterarbeit betreut hat und mir stets hilfsbereit zur Seite stand.

Literaturverzeichnis

[Cas13] Tommaso Castrovillari. Bachelorarbeit: Implementierung einer graphischen Benutzerschnittstelle für ein Programm zum automatischen Korrektheitsnachweis von Programmtransformationen in der funktionalen Programmiersprache Haskell, 2013. http://www.ki.informatik.uni-frankfurt.de/bachelor/abgeschlossen.html.

[Eid15] Philipp Eidam. Diplomarbeit: Automatische Laufzeitoptimierung von funktionalen Programmen in einer Kernsprache von Haskell, 2015. http://www.ki.informatik.uni-frankfurt.de/diplom/abgeschlossen.html.

[HF92] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *SIGPLAN Not.*, 27(5):1–52, May 1992. https://www.haskell.org/tutorial/; abgerufen am 29. Februar 2016.

[Mar] Simon Marlow. Happy - The Parser Generator for Haskell. https://www.haskell.org/happy/; abgerufen am 29. Februar 2016.

[Mar10] Simon Marlow. Haskell Language Report 2010, 2010. https://www.haskell.org/onlinereport/haskell2010/; abgerufen am 29. Februar 2016.

[Mar15] Simon Marlow. Fighting Spam with Haskell, 2015. https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/; abgerufen am 18. Januar 2016.

[OM16] Inc. O'Reilly Media. The History of Programming Languages, 2016. http://archive.oreilly.com/pub/a/oreilly//news/languageposter_0504.html; abgerufen am 18. Januar 2016.

[Sab15] David Sabel. Praktikum: Funktionale Programmierung [FP-PR], 2015. http://www.ki.informatik.uni-frankfurt.de/lehre/SS2015/FP-PR/main.html; abgerufen am 26. Januar 2016.

[Sch] Manfred Schmidt-Schauß. Skript zur Vorlesung "Grundlagen der Programmierung 2"(Sommersemester 2015). `http://www-stud.informatik.uni-frankfurt.de/~prg2/SS2015/skript/teil1/Kap-5-compiler.pdf`; abgerufen am 29. Februar 2016.

[SS15] Manfred Schmidt-Schauß and David Sabel. Improvements in a Functional Core Language with Call-by-Need operational Semantics. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 220–231, 2015.

[SSS08] Manfred Schmidt-Schauß, David Sabel, and Marko Schütz. Safety of Nöcker's strictness analysis. *J. Funct. Program.*, 18(4):503–551, 2008.

[SSS14] David Sabel and Manfred Schmidt-Schauß. Einführung in die funktionale Programmierung, 2014. `http://www.ki.informatik.uni-frankfurt.de/lehre/WS2014/EFP/skript/skript.pdf`; abgerufen am 26. Januar 2016.

[SSS15] Manfred Schmidt-Schauß and David Sabel. Sharing-aware Improvements in a Call-by-Need Functional Core Language. In Ralf Lämmel, editor, *Proceedings of IFL*, ACM, New York, NY, USA, 2015. ACM. to be published.

Abbildungsverzeichnis