Quellsprachentransformation von Pattern Matching in Haskell

Übersetzung von Pattern Matching und Case Completion

Malte Clement

Bachelorarbeit
Oktober 2019

Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
Prof. Dr. Michael Hanus, M.Sc. Finn Teegen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Die funktionale Programmiersprache Haskell bietet verschiedene Sprachkonstrukte an, um das Schreiben von Programmen zu vereinfachen. Diese Sprachkonstrukte erweitern den Sprachumfang, was dazu führt, dass das Verarbeiten von Programmen komplizierter werden kann.
Das Ziel dieser Arbeit ist es, Pattern Matching auf der linken Regelseite sowie Guards in andere, einfachere Sprachkonstrukte zu überführen. Dabei werden sowohl die Transformationen im Allgemeinen als auch die Implementierung und Anwendung der Transformationen beschrieben. Für die Übersetzung von Pattern Matching wird der Ansatz von Wadler "Efficient Compilation of Pattern-Matching" verfolgt. Dieser Ansatz wird dabei um eine Optimierung erweitert und leicht verändert, um die Linearität von Programmen zu erhalten. Die Guards hingegen werden anhand ihrer Semantik aus dem Haskell-Report übersetzt.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1. Motivation

Moderne Programmiersprachen wie Haskell stellen oft syntaktische Konstrukte bereit, die das Schreiben von Programmen vereinfachen oder diese besser lesbar machen. Man spricht dabei von syntaktischem Zucker. So erlaubt Haskell zum Beispiel partielle Definitionen, verschachteltes Pattern Matching auf der linken Regelseite, Guards und mehr. In Haskell lassen sich jedoch keine Eigenschaften automatisch beweisen oder verifizieren. Es stehen einem also zwei Optionen offen: Erstens, das umfangreiche Testen von Programmen, ohne Sicherheit, dass die gewünschten Eigenschaften erfüllt sind. Zweitens, das manuelle Beweisen der Eigenschaften. Dies kann gerade bei größeren Programmen oder mehreren Eigenschaften sehr aufwendig sein und bei einer Veränderung des Programms muss der Beweis erneut geführt werden. Dabei kann syntaktischer Zucker hinderlich sein, da seine semantische Bedeutung bekannt sein muss, um korrekte Beweise zu führen. Wenn man sich bei den Beweisen Zeit sparen will, dann kann es sich lohnen, einen Theorembeweiser oder Beweisassistenten zu verwenden. Doch diese stellen oft sehr strikte Anforderungen an die Programme, damit die Beweise korrekt sind. Beweisassistenten und Theorembeweiser nutzen meistens eine eigene Sprache, in die man sein Programm möglichst genau übersetzen muss. Bei dieser Übersetzung ist der syntaktische Zucker hinderlich, wenn zum Beispiel die Zielsprache die syntaktischen Konstrukte nicht unterstützt, kann es sehr schwer werden die Programme genau abzubilden. Aus diesem Grund sollten die Programme vor ihrer Übersetzung möglichst so vereinfacht werden, dass sie nur Syntax enthalten, die ein Äquivalent in der Zielsprache haben. Mit dem Ziel, Eigenschaften von Haskell-Programmen in Coq zu beweisen, beschäftigt sich diese Arbeit hauptsächlich mit der Transformation von komplexem Pattern Matching auf linken Regelseiten, dem Transformieren von Guards und dem Vervollständigen von `case`-Ausdrücken.

1.2. Gliederung

Die Arbeit ist in drei Kapitel unterteilt. Kapitel 2 beinhaltet die theoretischen Grundlagen dieser Arbeit. Darin werden die behandelten Sprachkonstrukte, Guards und Pattern und ihre formelle Übersetzung, eingeführt. Außerdem werden die Haskell Source Extentions angeschnitten, die eine Grundlage für die später beschriebene Implementierung darstellen. Kapitel 3 stellt die Struktur und Implementierung der Grundlagen vor. Der gesamte Trans-

formationsprozess ist dabei in mehrere Phasen unterteilt, die jeweils näher beschrieben sind. Außerdem wird zu jeder Phase beispielhaft gezeigt wie diese Verwendet werden kann. Kapitel 4 umfasst ein Fazit und den Ausblick. Im Fazit wird auf betstimmte Entscheidungen, die bei der Implementierung gemacht wurden, eingegangen. Es wird eine Alternative zum Algorithmus von Wadler vorgestellt und an einem Beispiel vorgerechnet. Der Ausblick umfasst einige Ergänzungen der Arbeit und Implementierung, die nicht in der Arbeit umgesetzt wurden, aber thematisch zur Arbeit passen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die nötig sind, um den Inhalt der Arbeit zu verstehen. Obwohl es nicht notwendig ist, kann es helfen, bereits Erfahrung in funktionaler Programmierung mit Haskell oder anderen funktionalen Sprachen zu haben. Zuerst werden die verschiedenen Arten von Pattern vorgestellt und wie Pattern Matching in Haskell verwendet werden kann. Dann werden Guards eingeführt und eine mögliche Übersetzung für diese definiert. Danach wird der Algorithmus von Wadler, eine Möglichkeit zur Übersetzung von Pattern Matching, vorgestellt. Abschließend werden die *Haskell Source Extentions*, die grundlegende Datenstrutkur zur Darstellung der Programme, angerissen.

2.1. Pattern Matching

Pattern Matching ist ein Sprachfeature von Haskell, welches einem ermöglicht, das Verhalten von Funktionen durch mehrere Gleichungen zu definieren. Dabei entspricht eine Gleichung jeweils dem Funktionsnamen, einer Liste von Pattern, wobei die Anzahl der Pattern für alle Gleichungen gleich sein muss, und einer rechten Regelseite.

$$
\begin{array}{lll}
f & p_{11} \ \ldots \ p_{1n} & = e_1 \\
 & \vdots \qquad\quad \vdots & \\
f & p_{m1} \ \ldots \ p_{mn} & = e_m
\end{array}
$$

In Haskell werden Pattern von oben nach unten und von links nach rechts gematcht. Wenn ein Pattern nicht passt, wird die nächste Gleichung überprüft und wenn die letzte Gleichung nicht matchen sollte, dann führt dies zu einem Laufzeitfehler. Im Sprachumfang von Haskell sind drei verschiedene Arten von Pattern definiert:

Variablenpattern
Diese entsprechen einer Variable und matchen mit jedem Pattern, wobei die Variable ans Pattern gebunden wird. Als Beispiel betrachten wir die Identitätsfunktion, die sich mit Hilfe von Variablenpattern wie folgt definieren lässt.

```
id :: a -> a
id x = x
```

Der Aufruf, der Funktion mit einem Argument führt dazu, dass der aktuelle Parameter an die Variable gebunden und dann zurückgegeben wird.

Wildcardpattern
Wildcardpattern, geschrieben als "_", sind den Variablenpattern sehr ähnlich. Sie matchen auch auf jedes Pattern, jedoch findet dabei keine Variablenbindung statt. Es ist Konvention Variablenpattern, die für die Berechnung nicht benötigt werden durch eine Wildcard zu ersetzen, um die Funktionen lesbarer zu gestalten. Mit Hilfe der Wildcard Pattern lässt sich die `const`-Funktion definieren.

```
const :: a -> b ->  a
const  x  _ = x
```

Da das zweite Argument nicht für die Berechnung genutzt wird, kann es als Wildcard geschrieben werden.

Konstruktorpattern
Konstruktorpattern bestehen aus einem k-stelligen Konstruktor und k Pattern. Ein Konstruktorpattern matcht genau dann, wenn sowohl der Konstruktor des ausgewerteten aktuellen Parameterer mit dem des Pattern gleich ist als auch alle Pattern des Konstruktors mit denen des ausgewerteten Parameters matchen. Mithilfe von Konstruktorpattern lassen sich viele Funktionen eleganter aufschreiben. Das gilt zum Beispiel für die Funktion `map`.

```
map :: (a -> b) -> [a] -> [b]
map _ [ ]      = [ ]
map f (x:xs) = f x : map f xs
```

Dabei ist ":" der zweistellige Listenkonstruktor in infix-Schreibweise. In diesem Fall ist die Funktion für jeden Konstruktor definiert, was aber in Haskell nicht notwendig ist. Einige Funktionen, wie zum Beispiel die `head`-Funktion, können partiell definiert werden.

```
head :: [a]  ->  a
head (x:_) = x
```

Dabei ist `head` für die leere Liste undefiniert, sodass ein Aufruf von `head` mit der leeren Liste zu einem Fehler führt.
Beim Aufruf einer Funktion mit Konstruktorpattern werde die aktuellen Parameter so weit ausgewertet, bis sie nicht mehr reduzierbar sind.

2.2. Guards

Guards sind eine Erweiterung von Pattern um eine boolsche Bedingung. Das bedeutet, dass sie sowohl direkt hinter den Pattern einer Funktionsdefinition als auch hinter den Pattern innerhalb eines `case`-Ausdrucks stehen können. Dabei kann jede Gleichung beliebig viele Guards haben.

Funktionsdefinitionen mit Guards haben folgende Form.

```
func p11 ... p1n | g11 = e11
                 |  ⋮
                 | g1k1 = e1k1
func p21 ... p2n | g21 = e21
                 |  ⋮
    ⋮            ⋮
func pm1 ... pmn | gm1 = em1
                 |  ⋮
```

Dabei stehen p_{ij} für das j-te Pattern der i-ten Regel, die g_{kl} entsprechen dem boolschen Ausdruck der k-ten Regel und des l-ten Guards und e_{kl} steht für die rechte Regelseite der k-ten Regel und des l-ten Guards. Allgmein gilt für Guards innerhalb von `case`-Ausdrücken Folgendes.

```
case v of {
      p1 | g11  → e11
         |       ⋮
         | g1n1 → e1n1
      ⋮
      pm | gm1  → em1
         |       ⋮
         | gmnm → emnm
      _          → e' }
```

Hierbei sind $p_1, \ldots, p_n$ Pattern und n_i die Anzahl der Guards von Pattern p_i. Die Auswertung von Guards erfolgt dabei ähnlich zu der Auswertung von Pattern. Eine Gleichung trifft zu, wenn sowohl alle Pattern passen als auch mindestens ein Guard zutrifft. Guards werden wie Pattern auch von oben nach unten getestet.

Als Beispiel für eine Funktion mit Guards und Pattern betrachten wir die `take`-Funktion, die die ersten n Elemente einer Liste zurückgibt.

```
take :: Int -> [a] ->  [a]
take _ [ ]                = []
take n (x:xs) | n <= 0    = []
              | otherwise = x : take (n-1) xs
```

Guards sind im Allgemeinen keine syntaktischer Zucker für Fallunterscheidungen, da eine Gleichung nur ausgewählt wird, wenn sowohl alle Pattern passen als auch mindestens ein Guard zutrifft. Eine Transformation zu einer Fallunterscheidung betrachtet nur die Pattern und nicht die Reihenfolge der Regeln. Wie Guards trotzdem transformiert werden können, wird im nächsten Abschnitt genauer betrachtet.

2.3. Übersetzung von Guards

Wie bereits im letzten Abschnitt erwähnt, lassen sich Guards nicht immer direkt in eine Fallunterscheidung übersetzen. Als Beispiel lässt sich folgende Definition der `not`-Funktion betrachten.

```
not x       | x = False
not False     = True
```

Wenn dies nun nur in Fallunterscheidungen übersetzt wird, sieht die Funktion wie folgt aus:

```
not' x      = if x then True else undefined
not' False  = True
```

In der Annahme, dass die Transformation korrekt ist, testet man die Funktion in der Konsole und stößt dabei auf folgenden Fehler.

```
> not  False = True
> not' False = undefined
```

Das Ersetzen der Guards durch eine Fallunterscheidung ist nicht für alle Funktionen korrekt, da die Reihenfolge der Regeln beim Pattern Matching für das Ergebnis der Funktion relevant ist.
Wenn Guards aber trotzdem korrekt übersetzt werden sollen, dann muss das Ausprobieren der anderen Regeln in der Transformation berücksichtigt werden.

Dafür findet man im Haskell Report Jones eine Definition, die beschreibt, wie man eine Kombination aus einem Pattern und beliebig vielen Guards übersetzt. Im Folgenden sind v, $e_1, \ldots, e_n$ und e' Ausdrücke, p ein Pattern und $g_1 \ldots g_n$ boolsche Ausdrücke.

```
case v of { p | g1 → e1 }
                ⋮
          | gn → en
          _      → e' }

= let y = e'
  in case v of {
           p → if g1 then e1 else ... else if gn then en else y
```

$\_ \rightarrow y$ }

Diese lässt sich nun für den Fall, dass mehrere Pattern vorkommen, verallgemeinern.
Sei n_i die Anzahl der Guards für das i-te Pattern.

```
case v of {
```
p_1 | g_{11} $\rightarrow e_{11}$

| $\vdots$

| g_{1n_1} $\rightarrow e_{1n_1}$

$\vdots$

p_m | g_{m1} $\rightarrow e_{m1}$

| $\vdots$

| g_{mn_m} $\rightarrow e_{mn_m}$

$\_ \rightarrow e'$ }

```
= let y = e'
```
c_1 = case p_1 of

$p_1 \rightarrow$ if g_{11} then e_{11} else ... else if g_{1n_1} then e_{1n_1} else c_2

$\_ \rightarrow c_2$

$\vdots$

c_m = case p_m of

$p_1 \rightarrow$ if g_{11} then e_{m1} else ... else if g_{mn_m} then e_{mn_m} else y

$\_ \rightarrow y$

in c_1

In dieser Transformation ist die Reihenfolge der Regeln korrekt abgebildet, da ein Fehler beim Pattern Matching oder der Fehlschlag aller Guards dazu führt, dass die nächste transformierte Regel getestet wird. Da aber auch Guards auf das Pattern Matching folgen können, muss die Regel für die Transformation noch um eine beliebige Anzahle von Pattern erweitert werden.

Im Folgenden sei g_{ij} der j-te Guard der i-ten Regel und e_{il} die rechte Regelseite des l-ten Guards der i-ten Regel.

func p_{11} ... p_{1n} |g_{11} = e_{11}

|$\vdots$

|g_{1k_1} = e_{1k_1}

func p_{21} ... p_{2n} |g_{21} = e_{21}

|$\vdots$

$\vdots$ $\vdots$

```
func p_m1 ... p_mn |g_m1 = e_m1
                   |⋮
func _    ... _        = e'
```

Durch die Transformation sieht die Funktion dann wie folgt aus.

```
func x_1 ... x_n =
    let y = e'
        c_1 = case x_1 of
          p_11 → case x_2 of
                    ⋱
              p_1(n−1) → case x_n of
                           p_1n → if g_11 then e_11 ... else if g_1k_1 then e_1k_1 else c_2
                               _ → c_2
                      _ → c_2
                 ⋰
            _ → c_2
        ⋮
        c_m = case x_1 of
          p_m1 → case x_2 of
                    ⋱
              p_m(n−1) → case x_n of
                           p_mn → if g_m1 then e_m1 ... else if g_mk_m then e_mk_m else y
                               _ → y
                      _ → y
                 ⋰
            _ → y
 in c_1
```

Dies entspricht semantisch dem Vorgehen vom Pattern Matching mit Guards. Von oben nach unten wird zuerst versucht von links nach rechts alle Pattern zu matchen, dann wird getestet, ob einer der Guards zutrifft. Wenn einer dieser Schritte fehlschlägt, wird zur nächsten Regel übergegangen.

2.4. Übersetzung von Pattern Matching

Ein Kernstück der Arbeit ist das Übersetzen von komplexem Pattern Matching auf der linken Regelseite. Dies wird gemacht, um sowohl das Pattern Matching zu vervollständigen als auch die Anzahl der Regeln auf eine Regel zu beschränken.

2.4.1. Algorithmus

Für die Übersetzung von Pattern Matching ist der Algorithmus von Wadler aus "Efficient Compilation of Pattern-Matching"Wadler (1987) implementiert. Da diesem Algorithmus bereits die Guard Eliminiation vorgeschaltet ist, kann bei der Übersetzung davon ausgegangen werden, dass die Funktionen keine Guards enthalten. Im weiteren ist die allgemeine Form einer Funktion folgende.

$$\begin{aligned} f\ \ p_{11} \ldots p_{1n}\ &= e_1 \\ \vdots \qquad\qquad\ &\ \ \vdots \\ f\ \ p_{m1} \ldots p_{mn} &= e_m \end{aligned}$$

,wobei p_{ij} für $i \in \{1, \ldots, m\}$ und $j \in \{1, \ldots, n\}$ entweder Konstruktor-oder Variablenpattern sind. Um Namenskonflikte zu vermeiden, werden zuerst n freie Variablen generiert.
Als Eingabe bekommt der Algorithmus nun die Liste der neu generierten Variablen $[x_1, \ldots, x_n]$, eine Liste von Paaren, die jeweils aus einer Liste von Pattern `[`$p_{i1}, \ldots, p_{in}$`]` ,wobei $i \in \{1, \ldots, m\}$ und einer rechten Regelseite bestehen, und einen Fehler `err`.
Der Fehler, der beim ersten Aufruf der Funktion übergeben wird, dient der Vervollständigung von partiellen Funktionen. Konstruktoren, für die eine Funktion nicht definiert ist, werden ergänzt und die rechte Seite des jeweiligen Falls entspricht dann `err`.
Auf die allgemeine Funktionsdefinition wird die `match`-Funktion angewendet, indem eine neue Regel mit der folgenden Form erzeugt wird.

$$f\ x_1 \ldots x_n = [\![\texttt{match [}x_1,\ldots,x_n\texttt{] [([}p_{11}\texttt{, ... ,}p_{1n}\texttt{],}e_1\texttt{), ... ,([}p_{m1}\texttt{, ... ,}p_{mn}\texttt{],}e_m\texttt{)] err}]\!]$$

Der `match`-Aufruf wird ausgeführt, was durch $[\![\ ]\!]$ gekennzeichnet ist, und das Ergebnis ist die neue rechte Regelseite.
Im Folgenden wird die für die `match`-Funktion folgende Schreibweise genutzt.

$$\texttt{match xs eqs err} \quad \text{mit} \quad \texttt{eqs} = \texttt{[(}ps_1\texttt{,}e_1\texttt{),...,(}ps_m\texttt{,}e_m\texttt{)]}$$

Dabei ist `xs` eine Liste von freien Variablen und ps_i mit $i \in \{1, \ldots, m\}$ eine Liste von Pattern. Nun lässt sich die `match`-Funktion durch vier Regeln definieren.

Alle Patternlisten fangen mit einer Variable an.
Wenn alle Listen von Pattern mit einer Variablen anfangen, dann wird in allen rechten Regelseiten die jeweilige Variable durch den Kopf der Variablenliste ersetzt.

$$\begin{aligned} [\![&\texttt{match (x:xs) [(}v_1 : ps_1\texttt{,}e_1\texttt{)} \\ &\qquad\qquad \vdots \\ &\qquad\texttt{, (}v_m : ps_m\texttt{,}e_m\texttt{)] err}]\!] \end{aligned}$$

$$
\begin{aligned}
= [\![&\texttt{match xs [} (ps_1,\ e_1[v_1 \mapsto \texttt{x}]) \\
&\qquad\qquad \vdots \\
&\qquad , \texttt{ } (ps_m,\ e_m[v_m \mapsto \texttt{x}])\texttt{] err}]\!]
\end{aligned}
$$

Hierbei bezeichnet $e[v \mapsto x]$ die Ersetzung von allen Vorkommen der Variable v durch x im Ausdruck e. Wenn die Liste von Paaren leer ist, dann lässt sich diese Regel trotzdem anwenden, da dann in den rechten Regelseiten nichts mehr verändert wird.

Alle Patternlisten fangen mit einem Konstruktor an.
Wenn alle Patternlisten mit einem Konstruktor anfangen, können wir einen `case`-Ausdruck erzeugen, bei dem wir über die verschiedenen Konstruktoren eine Fallunterscheidung machen. Um bei der Fallunterscheidung jeden Konstruktor genau einmal zu betrachten, werden die Paare nach den Konstruktoren sortiert. Die Sortierung muss stabil sein, damit die Semantik vom Pattern Matching korrekt abgebildet wird. (Pattern werden von oben noch unten bis zum ersten Treffer durchprobiert.) Eine Sortierung heißt stabil, wenn gleiche Objekte ihre interne Reihenfolge beibehalten.
Im Folgenden ist eqs_i mit $i \in \{1, \ldots, k\}$ die Liste, bei der alle Patternlisten mit C_i beginnen.

$eqs_i = \texttt{[((}C_i\ \ p_{i11} \ldots p_{i1a}\texttt{) :} ps_1,\ e_{i1}\texttt{),...,}\ \texttt{((}C_i\ \ p_{ib1} \ldots p_{iba}\texttt{) :} ps_b,\ e_{ib}\texttt{)]}$

Dabei ist a_i die Stelligkeit des jeweiligen Konstruktors und b die Anzahl der Paare in einer Gruppe.
Dann lässt sich die `match`-Funktion für einen Datentyp mit k Konstruktoren $C_1, \ldots, C_k$ und die sortierte Liste $(eqs_1 \texttt{++} \ \ldots \ \texttt{++}\ eqs_k)$ wie folgt definieren.

$$
\begin{aligned}
&[\![\texttt{match (x:xs) } eqs_1 \texttt{++} \ldots \texttt{++ } eqs_k \texttt{ err}]\!] \\
&= \texttt{case x of} \\
&\quad C_1\ \ x_{11},\ldots,x_{1a_1} \rightarrow [\![\texttt{match ([}x_{11},\ldots,x_{1a_1}\texttt{] ++ xs) } eqs_1^* \texttt{ err}]\!] \\
&\quad \vdots \\
&\quad C_k\ \ x_{k1},\ldots,x_{ka_k} \rightarrow [\![\texttt{match ([}x_{k1},\ldots,x_{ka_k}\texttt{] ++ xs) } eqs_k^* \texttt{ err}]\!]
\end{aligned}
$$

,wobei eqs_i^* durch $eqs_i^* = \texttt{[([}p_{i11} \ldots p_{i1a}\texttt{] ++ }ps_1,\ e_{i1}\texttt{), ... , ([}p_{ib1} \ldots p_{iba}\texttt{] ++ }ps_b,\ e_{ib}\texttt{)]}$ definiert ist.

Anmerkung: Wenn C_j für ein $j \in \{1, \ldots, k\}$ nicht in der gesamten Patternliste vorkommt, dann gilt $eqs_j = \texttt{[]}$.

Die erste Patternliste ist leer.
Diese Regel besteht aus zwei Teilen. Der Erste ist wichtig, damit der Algorithmus termi-

niert. Die Bedeutung der Zweiten wird relevant, wenn sowohl Variablenpattern als auch Konstruktorpattern am Beginn der Patternlisten stehen.

1. Eine Regel ist anwendbar.

   $[\![$`match [] [([], e) ... ] err`$]\!] =$ `e`

   Es kann sein, dass mehrere Regeln passen (dargestellt durch ...), dann wird die erste Passende genommen.

2. Keine Regel ist anwendbar.

   $[\![$`match [] [] err`$]\!] = [\![$`err`$]\!]$

   Wenn keine Regeln passen, dann wird die vorher mitgegebene (möglicherweise erweiterte) Fehlermeldung ausgewertet.

Sowohl Variablen als auch Konstruktoren liegen vor.
In diesem Fall muss wieder gruppiert werden, jedoch dieses mal nicht nach den Konstruktoren, sondern nach der Art des Pattern. Das Ergebniss der Gruppierung sind alternierende Gruppen, bei denen entweder jede Patternliste mit einer Variable oder jede Patternliste mit einem Konstruktor beginnt. Dabei soll die Reihenfolge beibehalten werden, sodass kein Paar, das mit einem Konstruktorpattern beginnt, über eins mit einer Variablenpattern hinweggezogen wird.

Ein Beispiel: ”Mississipi” würde bei einer Gruppierung nach Konsonanten und Vokalen wie folgt aussehen [”M”,”i”,”ss”,”i”,”ss”,”i”,”p”,”i”].

Sei $eqs = eqs_1 ++ \ldots ++ eqs_i$, wobei für alle $j \in \{1, \ldots, i-1\}$ gilt, wenn in eqs_j alle Patternlisten mit einem Variablenpattern beginnen, dann beginnen in eqs_{j+1} alle Patternlisten mit Konstruktorpattern. Dann kann man `match` wie folgt formalisieren.

$[\![$`match xs` eqs `err`$]\!]$
$= [\![$`match xs` eqs_1 `(match xs` eqs_2 `(...(match xs` eqs_i `err)...))`$]\!]$

2.4.2. Beispiel

Für die Übersetzung der `id`-Funktion werden die erste und die dritte Regel benötigt.

```
id :: a -> a
id x = x
```

Wenn der Algorithmus darauf angewendet wird, sieht das Ergebnis wie folgt aus.

```
id a0 = ⟦match [a0] [([x], x)] err⟧                Argumente für den Aufruf
      = ⟦match [] [([], x[x ↦ a0])] err⟧           Regel 1
      = ⟦match [] [([], a0)] err⟧                  Anwendung [x ↦ a0]
      =  a0                                        Regel 3.1
```

Der Algorithmus zur Übersetzung von Pattern Matching stellt auf Funktionen ohne Konstruktorpattern die Identität dar. Dies bedeuet, dass Funktionen, die keine Konstruktorpattern auf der linken Regelseite enthalten, vom Algorithmus nicht verändert werden.
Für die zweite Regel eignet sich die `not`-Funktion als Beispiel.

```
not :: Bool -> Bool
not True  = False
not False = True
```

Dann sieht der Aufruf von `match` wie folgt aus.

```
not a0 = ⟦match [a0] [([True], False),([False], True)] err⟧   Argumente für den Aufruf
       = case a0 of                                           Regel 2
           True  → ⟦match [] [([], False)] err⟧
           False → ⟦match [] [([], True)] err⟧
       = case a0 of                                           Regel 3.1 (zweimal)
           True  → False
           False → True
```

Als Beispiel für die vierte Regel kann man folgende Definition des logischen Oders benutzen.

```
or :: Bool -> Bool -> Bool
or True    x     = True
or x       True  = True
or False   False = False
```

Der Aufruf von `match` sieht dann wie folgt aus.

```
or a0  a1 = ⟦match [a0, a1] [ ([True, x ], True)
                             , ([x, True], True)
                             , ([False, False], False) ] err⟧
```

und durch Anwendung der vierten Regel dann folgendermaßen.

```
 = ⟦match [a0, a1] [([True, x), True)]
       (match [a0, a1] [([x, True), True)]
           (match [a0, a1] [([False, False), False)] err))⟧
```

2.4.3. Optimierung

Ein Problem des Algorithmus ist, dass es durch die vierte Regel dazu kommen kann, dass mehrfach auf dieselbe Variable gematcht wird. Als Beispiel dient die `zip`-Funktion. Sie ist nicht uniform, was bedeutet, dass die Reihenfolge der Regeln einen Einfluss auf das Ergebnis hat. Anders ausgedrückt ist eine Funktion uniform, wenn bei ihrer- Transformation nie die vierte Regel benutzt wird.

```
zip :: [a] -> [b] -> [(a,b)]
zip []     bs     = []
zip as     []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

Die transformierte Version sieht wie folgt aus.

```
 case x of
   []     -> []
   (z:zs) -> case y of
               []     -> []
               (q:qs) -> case x of
                           []     -> undefined
                           (t:ts) -> case y of
                                       []     -> undefined
                                       (u:us) -> (t,u) : zip ts us
```

Es wird zweimal auf `x` und `y` gematcht. Also lässt sich diese Funktion umschreiben, sodass nur einmal auf jeder Variable gematcht wird.

```
case x of
   []     -> []
   (z:zs) -> case y of
               []     -> []
               (q:qs) -> (z,q) : zip zs qs
```

Dabei werden bereits gematchte Pattern und die Variablen, an die sie gebunden sind, gespeichert. Wenn dann noch einmal auf dieses Pattern gematcht wird, wird der `case`-Ausdruck durch die rechte Regelseite des `case`-Ausdrucks für das Pattern ersetzt.
Definiere also eine Umgebung U, die eine Menge von Paaren mit je einer Variable und den daran gebundenen Konstruktor enthält. Um also einen einen Ausdruck e zu optimieren, definiert man eine Funktion `opt`, die einen Ausdruck und eine Umgebung als Argumente erhält und sich wie folgt verhält.

- ▷ e ist kein `case`-Ausdruck.
  Wenn e kein `case`-Ausdruck ist, dann optimiere alle Teilausdrücke rekursiv.

▷ e ist ein `case`-Ausdruck.
Für einen Datentyp mit $n \in \mathbb{N}$ Konstruktoren, wobei k_i mit $i \leqslant n$ die Stelligkeit des i-ten Konstruktors ist, sieht ein `case`-Ausdruck wie folgt aus.

$$\begin{array}{l}\texttt{case } v \texttt{ of } \{ \\ \quad C_1 \;\; p_{11} \ldots p_{1k_1} \rightarrow e_1 \\ \quad \vdots \\ \quad C_n \;\; p_{n1} \ldots p_{nk_n} \rightarrow e_n \\ \}\end{array}$$

Für den Fall, dass v keine Variable ist, optimiere v und dann optimiere alle rechten Regelseiten. Sei U die aktuelle Umgebung, dann lassen sich zwei Fälle unterscheiden:

1. $v \notin U$
   Der aktuelle `case`-Ausdruck lässt sich nicht optimieren, aber rekursiv die rechten Regelseiten.

   $$\begin{array}{l}\texttt{case } v \texttt{ of } \{ \\ \quad C_1 \;\; p_{11} \ldots p_{1k_1} \rightarrow [\![\texttt{opt } e_1 \;\; U \cup \{(v, \;\; C_1 \;\; p_{11} \ldots p_{1k_1})\}]\!] \\ \quad \vdots \\ \quad C_n \;\; p_{n1} \ldots p_{nk_n} \rightarrow [\![\texttt{opt } e_n \;\; U \cup \{(v, \;\; C_n \;\; p_{n1} \ldots p_{nk_n})\}]\!] \\ \}\end{array}$$

2. $v \in U$
   Wenn ein Tupel $(v, C_i \; q_{ik_1} \ldots q_{ik_i})$ exisitert, kann der `case`-Ausdruck durch

   $$\texttt{opt } (e_i[p_{i1} \mapsto q_{i1}, \ldots, p_{ik_i} \mapsto q_{ik_i}]) \;\; U$$

   ersetzt werden. Dabei steht $e[a \mapsto b]$ für eine Umbenennung von a durch b im gesamten Ausdruck e. Die weitere Optimierung des gesamten Ausdrucks ist notwendig, da weitere `case`-Ausdrücke enthalten sein können.

2.5. Haskell Source Extentions

Die gesamte Implementierung basiert auf den Haskell-Src-Extentions. Dieses cabal-Paket stellt einen Parser und Lexer bereit, der Haskell-Programme in eine abstrakte Syntaxdarstellung überführt. Zum Beispiel sieht folgende Definition der `head`-Funktion

```
head (x:xs) = x
```

in der AST-Representation wie folgt aus. (Die Kommentare gehören dabei nicht zur Representation.)

```
FunBind ()
  [Match ()
   (Ident () "head")
   [PParen ()                 -- Pattern in Klammern
     (PInfixApp ()            -- Infix-Konstruktor
       (PVar ()               -- Variablenpattern
         (Ident () "x")
       )
       (Special ()            -- spezieller Haskell-Typ
         (Cons ())            -- Listenkonstruktor
       )
       (PVar ()
         (Ident () "xs")
       )
     )
   ]
   (UnGuardedRhs ()           -- rechte Regelseite
     (Var ()                  -- Variable auf der rechten Regelseite
       (UnQual ()
         (Ident () "x")
        )
      )
    )
    Nothing
  ]
```

Jedes valide Haskell-Programm hat eine Darstellung als Syntaxbaum, bei dem jeder Teilausdruck in einen Baum aus Konstruktoren übersetzt wird. Dabei bieten die Haskell Source Extentions einen Lexer und Parser an, um Module in ihre abstrakte Syntaxrepresentation zu überführen. Der Algorithmus von Wadler arbeitet hauptsächlich auf Pattern und Ausdrücken. Diese werden dabei durch folgende Typen repräsentiert.

```
data Pat l = PVar l (Name l)              -- Variable pattern
           | PApp l (Qname l)  [Pat l]    -- Constructor pattern with argument patterns
           | PWildcard l                  -- Wildcard pattern
           | ⋮
```

Der Typ `Pat l` beinhaltet neben den in Abschnitt 2.1 beschriebenen Pattern noch weitere. Diese werden in der Übersetzung aber nicht berücksichtigt.

```
data Exp l = Var l (QName l)                        -- Variable expression
           | Con l (QName l)                        -- Data constructor
           | Lit l (Literal l)                      -- Constants
```

```
  | App l (Exp l) (Exp l)                -- Application
  | Lambda l [Pat l] (Exp l)            -- Lambda expression
  | Let l (Binds l) (Exp l)             -- Let binding in exp
  | If l (Exp l) (Exp l) (Exp l)        -- If-then-else expression
  | Case l (Exp l) [Alt l]              -- Case expression
  | ⋮
```

Der Typ `Exp l` hat Konstruktoren für alle Sprachkonstrukte, die auf der rechten Regelseite vorkommen können. Die Implementierung benutzt zum einen die Konstruktoren für Variablen, Konstruktoren, Funktionsanwendungen und Konstanten, zum anderen Konstruktoren für gängige Sprachfeatures wie Lambda-,`case`- und `let`-Ausdrücke. Die meisten nicht unterstützten Konstruktoren werden nicht unterstützt, weil sie nicht mit Hilfe des Algorithmus übersetzt werden können. Das `l` in den Typen und Konstruktoren steht für die Span-Information, die aber für den Algorithmus nicht benötigt wird. Span-Informationen sind Positionsangaben für jedes Konstrukt innerhalb eines Programms

Kapitel 3

Struktur und Implementierung

In diesem Kapitel wird die Struktur und Implementierung der theoretischen Grundlagen aus Kapitel 2 beschrieben. Dabei wird zuerst die Struktur des Transformationsprozesses verdeutlicht und danach ein Überblick über die Phasen und einzelne Implementierungsaspekte gegeben. Die Phasen werden in der Reihenfolge vorgestellt in der sie später in der Implementierung genutzt werden. Zuerst der Algorithmus und die Zustandsmonade, die keine fester Position in der Reihenfolge haben, da sie in mehreren Phasen eingesetzt werden. Danach die *Guard Elimination* und die *Case Completion*. Letztere wird sowohl theoretisch als auch an einem Beispiel erklärt. Zum Abschluss wird dann die *Pattern Match Compilation* erklärt und einmal auf ein beispielhaftes Modul angewandt.

3.1. Aufbau

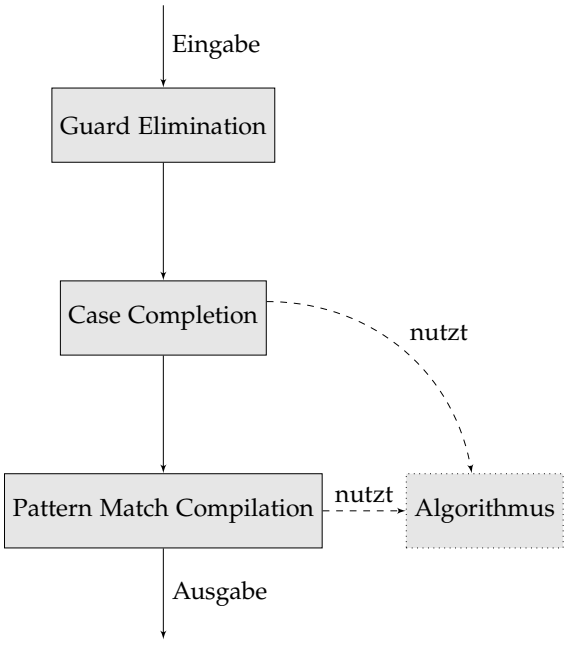

Abbildung 3.1. Ablauf der Transformationen

Der gesamte Transformationsprozess ist in mehrere Phasen unterteilt. Dabei entspricht der Algorithmus der Definition aus Abschnitt 2.4. Zuerst werden mithilfe der *Guard Elimination* alle Guards in Fallunterscheidungen und `case`-Ausdrücke übersetzt, damit die `case`-Ausdrücke in der nächsten Phase, der *Case Completion*, zu vollständigen Ausdrücken transformiert werden können. Die letzte Phase, die *Pattern Match Compilation*, verwendet den Algorithmus um Pattern Matching auf linken Regelseiten in vollständige `case`-Ausdrücke auf rechten Regelseiten zu transformieren.
Alle Phasen funktionieren zwar unabhängig voneinander, aber die Reihenfolge innerhalb des Ablaufs ist aufeinander abgestimmt. So dürfen Funktionen vor Verwendung des Algorithmus keine Guards beinhalten, weswegen die *Guard Elimination* vorher ausgeführt wird. Außerdem wird die *Case Completion* vor der *Pattern Match Compilation* ausgeführt, damit die von der *Pattern Match Compilation* generierten `case`-Ausdrücke, die bereits vollständig sind, nicht nochmal von der *Case Completion* überprüft werden. Da *Guard Elimination* `case`-Ausdrücke erzeugt, die unvollständig sind, sollte diese Phase vor den anderen liegen.
Damit das generierte Programm semantisch und syntaktisch korrekt ist und der Transformationsprozess fehlerfrei abläuft, müssen einige Bedingungen erfüllt sein.

- **Syntaktische Korrektheit**
  Die Programme müssen syntaktisch korrekt sein. Teil davon ist die Typkorrektheit, die aber nicht überprüft wird.

- **Keine Spracherweiterungen**
  Das Program darf keine Spracherweiterungen voraussetzen. Es ist aber möglich Transformationen für bestimmte Spracherweiterungen selbst zu implementieren.

- **Keine `do`-Notation**
  Das Program darf keine `do`-Notation enthalten.

- **Keine lokale Funktionen**
  Das Programm muss ohne lokale Funktionen definiert sein.

- **Kein Pattern Matching auf Zahlen**
  Das Programm darf kein Pattern Matching auf Zahlen enthalten.

- **Einschränkung von `let`-Ausdrücken**
  Es dürfen keine rekursiven `let`-Ausdrücke in den Programmen enthalten sein. Außerdem darf auf der linken Seite eines `let`-Ausdrucks kein Pattern Matching stattfinden.

3.2. Zustandsmonade

Da in jeder Phase frische, nicht im ursprünglichen Programm vorkommende, Variablen benötigt werden, wird der laufenden Index der Variablen in einem Zustand gespeichert.

```
data PMState = PMState { nextId     :: Int
                      , constrMap  :: [(String, [Constructor])]
                      , matchedPat :: [(S.Exp (), S.Pat () )]}
```

Zustätzlich zu dem aktuellen Identifikator für generierte Variablen enthält der Zustand eine Zuordnung von Datentypnamen auf zugehörige Konstruktoren. Dabei ist

```
type Constructor = (String, Int)
```

ein Typsynonym für Namen und Stelligkeit eines Konstruktors.
Außerdem enthält der Zustand eine Liste, *matched pat*, von Variablen und Pattern, wobei die Variablen an die Pattern gebunden sind. Dabei ist die Variable intern als Ausdruck gespeichert und mit `S` qualifiziert. Diese Liste wird für die Optimierung von `case`-Ausdrücken benötigt, die in Abschnitt 2.4.3 beschrieben wird. Zusätzlich enthält der Zustand auch einige Wahrheitswerte, die der Übersichtlichkeit halber nicht aufgeführt sind, mit denen der Programmablauf verändert werden kann, um zum Beispiel detaillierte Debug-Informationen zu erhalten.

Da für den Algorithmus und die *Case Completion* alle lokal definierten Konstruktoren benötigt werden, werden diese vor Beginn der Transformation aufgesammelt und in den Zustand geschrieben. Dabei müssen Listen, Tupel (bis zu einer Länge von sieben) und Unit nicht selbst definiert werden, da diese immer im Zustand gespeichert sind.
Der Zustand wird in allen Phasen benötigt, da die *Guard Elimination* und der Algorithmus Variablen generieren und letzterer zusätzlich noch die Konstruktoren benötigt.

3.3. Guard Elimination

In der ersten Phase, der *Guard Elimination*, werden alle Guards in Funktionen, wie in Abschnitt 2.3, durch eine Kombination von `case`-Ausdrücken und Fallunterscheidungen ersetzt. Allerdings ist die *Guard Elimination* auf Funktionen ohne Guards nicht die syntaktische Identität. Dies lässt sich an folgendem Beispiel zeigen:

```
id :: a -> a
id x = x
```

Diese Definition der `id`-Funktion wird von der *Guard Elimination* zu

```
id :: a -> a
id a0
  = let a2 = undefined
        a1 = case a0 of
                x -> x
                _ -> a2
     in a1
```

übersetzt und selbst durch ein *Let Inlining*

```
id :: a -> a
id a0 = case a0 of
          x -> x
          _ -> undefined
```

entspricht die transformierte Version nicht der ursprünglichen Definition. Die Semantik der Funktionen bleibt zwar gleich, aber die *Guard Elimination* sollte nur auf Funktionen, die Guards enthalten, angewendet werden. Deswegen wird in der Implementierung getestet, ob in einer Funktion Guards vorkommen und nur dann wird die *Guard Elimination* angewandt. Dabei ist das im Beispiel angewendete *Let Inlining* optional, da auch dies die Semantik nicht verändert.

3.4. Case Completion

In der nächste Phase, der *Case Completion*, werden die in Funktionen enthaltenen `case`-Ausdrücke vervollständigt. Um `case`-Ausdrücke zu vervollständigen, gibt es zwei verschieden Ansätze. Der erste, triviale Ansatz zur *Case Completion* ist es, jeden `case`-Ausdruck, der kein Wildcard- oder Variablenpattern enthält, um ein Wildcardpattern mit einem Fehlerfall zu erweitern. Der zweite Ansatz ist eine Anwendung des Algorithmus auf jeden `case`-Ausdruck, um die Eigenschaft zu nutzen, dass dieser vollständige `case`-Ausdrücke erzeugt.

3.4.1. Definition

Seien $C_1, \ldots, C_k$ Konstruktoren eines Datentyps mit $l > k$ Konstruktoren. Dabei steht a_i für die Stelligkeit von Konstruktor C_i. Dann lässt sich ein `case`-Ausdruck folgender Gestalt

$$
\begin{array}{l}
\texttt{case } x \texttt{ of} \\
\quad C_1 \;\; p_{11}, \ldots, p_{1a_1} \rightarrow e_1 \\
\quad \vdots \\
\quad C_k \;\; p_{k1}, \ldots, p_{ka_k} \rightarrow e_k
\end{array}
$$

umschreiben zu

$$\texttt{case } x \texttt{ of}$$
$$y \to [\![\texttt{match } [y]\ [([C_i\ \ p_{11} \ldots p_{1a_1}]\ ,\ e_1),\ \ldots\ ,([C_i\ \ p_{k1} \ldots p_{ka_k}]\ ,\ e_k)]\ \texttt{err}]\!]$$

$$= \texttt{case } x \texttt{ of}$$
$$y \to \texttt{case } y \texttt{ of}$$
$$C_1\ \ p_{11}, \ldots, p_{1a_1} \to e'_1$$
$$\vdots$$
$$C_l\ \ p_{l1}, \ldots, p_{la_l} \to e'_l$$

,wobei y und x_{ij} für $i \in \{1, \ldots, l\}$ und $j \in a_1, \ldots, a_l$ frische Variablen sind.
Die e'_i entsprechen dabei *err*, wenn $C_i \notin \{C_1, \ldots, C_k\}$, ansonsten den rekursiv vervollständigten rechten Regelseiten.

Der generierte `case`-Ausdruck kann dann noch optimiert werden zu

$$\texttt{case } x \texttt{ of}$$
$$C_1\ \ p_{11}, \ldots, p_{1a_1} \to e'_1$$
$$\vdots$$
$$C_l\ \ p_{l1}, \ldots, p_{la_l} \to e'_l$$

Da die Variable x an y gebunden wird und auf y eine Fallunterscheidung gemacht wird, entspricht dies der Fallunterscheidung über x.

3.4.2. Anwendung

Die Anwendung der *Case Completion* auf eine Funktion entspricht dabei einem rekursiven Aufruf einer Funktion `completeCase` über den Syntaxbaum für jede Regel. Also wird für jede Regel die rechte Regelseite betrachtet und dann wird in der Funktion eine Fallunterscheidung gemacht.

- **`case`-Ausdruck**
  Wenn der Ausdruck ein `case`-Ausdruck ist, wird dieser nach der Definition mit einem Aufruf von `match` vervollständigt.

- **Funktionsanwendung**
  Bei einem Ausdruck der Form $f\ x$, wobei f und x Ausdrücke sind, werden beide Ausdrücke rekursiv mit einem Aufruf von `completeCase` vervollständigt.

Allerdings gibt es zwei Sonderfälle, die bei der *Case Completion* beachtet werden müssen:

- **`let`-Ausdrücke**
  `let`-Ausdrücke können für die Vervollständigung problematisch sein, da auf der linken

Seite ein Konstruktorpattern stehen kann. Dies führt zu einem Fehler, wenn der Ausdruck nicht zu dem Pattern reduziert werden kann. Um `let`-Ausdrücke zu transformieren, benötigt man Selektorfunktionen für den gematchten Konstruktor. Wie die Transformation um die Vervollständigung von `let`-Ausdrücken erweitert werden kann, wird im **??** beschrieben.

▻ **anonyme Funktionen**
In Haskell können anonyme Funktionen folgender Form vorkommen.

`\` $(C\ p_1 \ldots p_k) \rightarrow e$

Dabei sei C ein k-stelliger Konstruktor und $p_1 \ldots p_k$ Pattern. Diese Funktion lässt sich dann mittels `match`-Funktion zu

`\ x →` `match [x] [(`$[C\ p_1 \ldots p_k]$`,` e'`)] err`

vervollständigen, wobei x eine freie Variable, e' die rekursiv vervollständigte rechte Regelseite und `err` wie bisher ein definierter Standardfehler ist.

3.4.3. Beispiel

Als Beispiel für eine Anwendung ohne Sonderfall kann man folgende Definition der `head`-Funktion betrachten.

```
head :: [a] -> a
head x = case x of
           (y:ys) -> y
```

Die Definition enthält einen unvollständigen `case`-Ausdruck, somit ist die *Case Completion* anwendbar.

```
head :: [a] -> a
head x = case x of
           a0 -> ⟦match [a0] [((y:ys,y)] err⟧
```

Dabei stehen die doppelten, eckigen Klammern dafür, dass der Inhalt noch ausgewertet werden muss. Nach der Anwendung sieht der `case`-Ausdruck dann wie folgt aus.

```
head :: [a] -> a
head x = case x of
           a0 -> case a0 of
                   []     -> undefined
                   (a1:a2) -> a1
```

das Endergebniss lässt sich, wie beschrieben, dann noch zu

```
head :: [a] -> a
head x = case x of
           []      -> undefined
           (a1:a2) -> a1
```

optimieren.

3.5. Pattern Match Compilation

Die letzte Phase, die *Pattern Match Compilation*, ist die Anwendung des Algorithmus auf alle Funktionen, die durch Konstruktorpattern definiert sind. Dabei wird aus jeder Regel ein Paar aus den Pattern und dem Ausdruck der rechten Regelseite geformt. Dann werden Variablen entsprechend der Stelligkeit der Funktion generiert. Die Paare und Variablen werden anschließend der `match`-Funktion übergeben.

Auf folgendes Modul wird die *Pattern Match Compilation* also wie folgt angewandt.

```
module Example where

id :: a -> a
id x = x

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : xs

head :: [a] -> a
head (x:xs) = x
```

Die `match`-Funktion wird nur auf `map` und `head` angewandt, weil nur diese Funktionen durch Muster definiert sind.

```
module Example where

id :: a -> a
id x = x

map :: (a -> b) -> [a] -> [b]
map a0 a1 = ⟦match [a0,a1] [([f, []], []), ([f, (x:xs)], f x : xs)] err⟧

head :: [a] -> a
head a2 = ⟦match [a2] [([(x:xs)], x)] err⟧
```

Kapitel 4

Fazit und Ausblick

In diesem Kapitel werden die während der Implementierung getroffenen Entscheidungen und Alternativen näher betrachtet und begründet. Außerdem werden mögliche, nicht implementierte Erweiterungen der Arbeit vorgestellt.

4.1. Fazit

Im Rahmen dieser Arbeit wurde ein Tool implementiert und erarbeitet, das Haskell-Programme einliest und dann in Haskell-Programme umformt, in denen keine Funktion Guards enthält, durch Muster auf der linken Regelseite oder partiell definiert ist. Dabei wird sowohl Partialität auf Regelebene als auch innerhalb von `case`-Ausdrücken transformiert.
Die Funktionalitäten der verschiedenen Phasen *Guard Elimination*, *Case Completion*, *Pattern Match Compilation* werden als Library in einem cabal-Paket bereitgestellt.

4.1.1. Auswahl des Algorithmus

Zum Übersetzen des Pattern Matching gibt es mehrere Ansätze. Zum einen gibt es den Algorithmus von Wadler zur Übersetzung von Pattern Matching, der in der Arbeit auch implementiert und in Abschnitt 2.4 bereits beschrieben wurde. Außerdem gibt es einen ähnlichen Ansatz von Lennart Augustsson, der in "Compiling pattern matching"Augustsson (1985) beschrieben wird. Der Ansatz ist auch über mehrere Regeln definiert, aber durch die Nutzung von `gotos` werden die `case`-Ausdrücke bereits optimieren und doppeltes Pattern Matching vermieden. Der dritte Ansatz war die Übersetzung von Pattern Matching mittels endlicher Automaten von Mikael Pettersson Pettersson (1992).

Bei der Auswahl wurde der zweite Ansatz nicht weiter verfolgt, da dieser auf `gotos` basiert und die Verwendung dieser als schlechter Programmierstil empfunden wird. Dijkstra (2002) Außerdem wurde als Programmiersprache für das Tool Haskell ausgewählt und die Sprache dieses Sprachfeature nicht anbietet.

Ohne den Algorithmus formal einzuführen, wird er im Folgenden einmal benutzt, um die `zip`-Funktion als Beispiel zu übersetzen. Die Übersetzung der `zip`-Funktion durch den Algorithmus von Wadler ist aus Platzgründen im Anhang zu finden. Der Algorithmus besteht aus vier sequentiellen Phasen. Zuerst werden alle Pattern so umbenannt, dass sie ihre Position in der Regel repräsentieren. Danach wird mit Hilfe einer Funktion namens `match` ein

deterministischer endlicher Automat (DEA) konstruiert. Dieser Automat wird in der nächsten Phase noch optimiert, indem gleiche Zustände zusammengeführt werden. Als Letztes wird aus dem Automaten Programmcode generiert.

```
zip :: [a] -> [b] -> [(a,b)]
zip [] y  = []     = []
zip x  []          = []
zip  (x':xs') (y':ys') = (x',y') : zip xs' ys'
```

Schritt 1: Umbenennung Zuerst werden alle Gleichungen und Pattern umbenannt. Dabei steht die Schreibweise $es[a/x]$ für die Bindung von dem Pattern an Position a an die Variable x im Ausdruck e.
Nach der Umbenennung erhält man eine Matrix von Pattern und eine Spalte von Endzuständen, wobei qi der i-te Endzustand ist.

$$\begin{pmatrix} xs = \text{nil} & ys = \_ \\ xs = \_ & ys = \text{nil} \\ xs = \text{cons}(xs.2.1 = \_\,,\ xs.2.2 = \_) & ys = \text{cons}(ys.2.1 = \_\,,\ ys.2.2 = \_) \end{pmatrix}$$

$$\begin{pmatrix} q1 = [] \\ q2 = [] \\ q3 = (x',\ y')\ :\ zip\ xs'\ ys'[xs.2.1/x',\ xs.2.2/xs',\ ys.2.1/y',\ ys.2.2/ys'] \end{pmatrix}$$

Schritt 2: Konstruktion des DEA Zuerst wird `match` auf die initiale Matrix und Endzustände angewendet. Da in der obersten Zeile der Matrix ein Konstruktor vorkommt, wird die Mischungs-Regel verwendet, die einen Zustand `q0` generiert

```
q0 : case xs of
       nil                  -> match1
       cons(xs.2.1, xs.2.2) -> match2
```

mit

```
match1 : {ys = _  } {q1}
         {ys = nil} {q2}

match2 : {xs.2.1 = _, xs.2.2 = _, ys = nil                          } {q2}
         {xs.2.1 = _, xs.2.2 = _, ys = cons(ys.2.1 = _, ys.2.2 = _)} {q3}
```

Durch die Variablen-Regel kann `match1` direkt zu `q1` reduziert werden.
Für `match2` ist wieder die Mischungs-Regel anwendbar. Was einen neuen Zustand `q4` generiert mit

```
q4 : case ys of
       nil                  -> match3
       cons(ys.2.1, ys.2.2) -> match4
```

mit

```
match3 : {xs.2.1 = _, xs.2.2 = _                        }{q2}

match4 : {ys.2.1 = _, ys.2.2 = _, xs.2.1 = _, xs.2.2 = _}{q3}
```

Sowohl auf `match3` als auch `match4` lässt sich die Variablen-Regel anwenden, sodass diese direkt zu `q2` und `q3` reduziert werden. Dann sieht der endgültige Automat wie folgt aus

```
q0 :: case xs of
        nil                 -> q1
        cons(xs.2.1,xs.2.2) -> q4

q1 :: nil
q2 :: nil

q3 :: (x',y') : zip xs' ys'[xs.2.1/x', xs.2.2/xs', ys.2.1/y', ys.2.2/ys']

q4 :: case ys of
        nil                 -> q2
        cons(ys.2.1,ys.2.2) -> q3
```

Schritt 3: Optimierung Die Endzustände `q1` und `q2` sind äquivalent und können somit zusammengefasst werden.

Schritt 4: Code-Generierung Nun werden die Endzustände in eine Funktionsdefinition übersetzt

```haskell
zip xs ys = case xs of
              []     -> []
              (x:xs) -> case ys of
                          []     -> []
                          (y:ys) -> (x,y) : zip xs ys
```

Letzten Endes ist die Auswahl auf den Algorithmus von Wadler gefallen, da die Implementierung auf Datenstrukturen zurückgreift, die von Haskell angeboten werden und somit keine weiteren Implementierungen von Datenstrukturen wie zum Beispiel Automaten notwendig sind. Außerdem generiert der Algorithmus vollständige `case`-Ausdrücke, selbst wenn die Definitionen partiell sind, weswegen dieser auch für die *Case Completion* benutzt werden kann. Auch der Nachteil, dass es bei nicht uniformen Funktionen zu doppeltem Pattern Matching kommt, kann, wie bereits beschrieben, durch eine Optimierung ausgeglichen werden.
Dazu kommt, dass die Implementierung des gewählten Ansatzes in einer Sprache wie Haskell gut umsetzbar ist, da der Algorithmus aus einer einzelnen Funktion besteht, die über verschiedene Fälle definiert ist. Die Implementierung kann somit sehr nah an der Definition

gehalten werden, was die Verständlichkeit des produzierten Codes verbessert.

4.1.2. Guard Elimination als Alternative zur Pattern Match Compilation

Wie im Kapitel zur Struktur bereits beschrieben ist, wird in der *Pattern Match Compilation* der Algorithmus auf alle Funktionsdefinitionen, die durch Muster definiert sind, angewendet. Dies wird gemacht um die Funktionen zu vervollständigen und durch das explizite Pattern Matching auf der rechten Seite alle Funktionen mit je einer Regel zu definieren. Da die Guard Elimination auch das Pattern Matching in `case`-Ausdrücke übersetzt und somit die Anzahl der Regeln auf eins reduziert, stellt die *Guard Elimination* eine Alternative zur *Pattern Match Compilation* dar. Dazu muss die *Guard Elimination* auf alle Funktionen angewendet werden, die mittels Guards, Mustern oder mehrerer Regeln definiert sind. Dadurch, dass die `case`-Ausdrücke semantisch korrekt übersetzt würden, wäre die *Pattern Matching Compilation* nicht mehr benötigt und der Algorithmus würde dann nur für die *Case Completion* benutzt. Die `not`-Funktion würde dann von der *Guard Elimination* wie folgt übersetzt werden.

```
not :: Bool -> Bool
not True  = False
not False = True
```

wird zu

```
not x = let a0 = undefined
            a1 = case x of
                   True  -> False
                   _     -> a2
            a2 = case x of
                   False -> True
                   _     -> a0
```

4.1.3. Erhaltung der Linearität von Funktion

Eine implementierte Erweiterung des Algorithmus ist es, sobald ein `case`-Ausdruck generiert wird, in den Pattern-Ausdruck-Paaren alle Vorkommen der Variable durch das jeweilige Konstruktorpattern zu ersetzen.

Dadurch verändert sich die zweite Regel des Algorithmus.

`match (x:xs)` eqs_1 `++ ... ++` eqs_k `err`

$=$ `case` x `of`
$C_1\ x_{11},\ldots,x_{1a_1} \rightarrow [\![$ `match (` $[x_{11},\ldots,x_{1a_1}]$ `++` xs `)` eqs_1^* err $]\!]$ $[x \mapsto (C_1\ x_{11},\ldots,x_{1a_1})]$
$\vdots$

$$C_k\ \ x_{k1},\ldots,x_{ka_k} \rightarrow [\![\ \texttt{match}\ (\texttt{[}x_{k1},\ldots,x_{ka_k}\texttt{]} +\!\!+ xs)\ \ eqs_k^*\ \ err\ ]\!]\ \texttt{[}x \mapsto (C_k\ \ x_{k1},\ldots,x_{ka_k})\texttt{]}$$

Das führt dazu, dass lineare Funktionen auch nach der Transformation ihre Linearität beibehalten. In Haskell stellen nicht lineare Funktionen zwar kein Problem dar, aber wenn die Definition einer Funktion linear ist, sollte dies auch auf nach der Transformation erhalten bleiben. Gerade mit der Übersetzung von Haskell-Programmen in andere Sprachen im Hinterkopf ist dies erwünscht, um etwaige Effekte nicht zu duplizieren.
Wie die Transformation angepasst wird, lässt sich an folgendem Beispiel illustrieren.
Für eine Queue-Implementierung durch ein Paar aus Listen

```
type QueueI a = ([a],[a])
```

kann man die Funktion `flipQ` wie folgt implementieren.

```
flipQ :: QueueI a -> QueueI a
flipQ ([],b) = (reverse b,[])
flipQ q      = q
```

Dabei ist die zweite Regel für das Beispiel interessant, da diese eine partielle Identität für den `QueueI`-Datentyp darstellt. Bei der Anwendung der Definition der `match`-Funktion sieht das Ergebenis wie folgt aus

```
flipQ :: QueueI a -> QueueI a
flipQ a20
  = case a20 of
      (a21, a22) -> case a21 of
                      []          -> (reverse a22, [])
                      (a25 : a26) -> a20
```

Diese übersetzte Funktion ist nicht linear, da `a20` auf der rechten Seite der Funktion mehr als einmal vorkommt. Damit die Funktion wieder linear wird, kann `a20` (innerhalb des ersten cases) auf `(a21,a22)` und (`a21` innerhalb des zweiten cases) auf `(a25 : a26)` abgebildet werden. Dann erhält man folgende Funktion.

```
flipQ :: QueueI a -> QueueI a
flipQ a20
  = case a20 of
      (a21, a22) -> case a21 of
                      []          -> (reverse a22, [])
                      (a25 : a26) -> (a25 : a26, a22)
```

Diese Funktion ist linear und semantisch gleichbedeutend zur ursprünglichen Funktion. Der Algorithmus kann aber keine Funktionen, die bereits als nicht linear definiert sind, zu linearen Funktionen transformieren.

4.2. Ausblick

Dieser Abschnitt behandelt Inhalte, die thematisch zur Arbeit passen oder sinnvolle Erweiterungen darstellen, aber nicht Teil der Arbeit sind. Dabei sind das *Let Inlining* und die Übersetzung von Fallunterscheidungen weitere Phasen, wohingegen das Pattern Matching und die *Case Completion* für `let`-Ausdrücke eine Erweiterung der bisherigen Funktionalität darstellen.
Das Tool ist strukturell so gehalten, dass zukünftige Erweiterungen, zum Beispiel weitere Phasen oder Unterstützung eines größeren Sprachumfangs, einfach hinzuzufügen sind.

Let Inlining
Beim *Let Inlining* wird für Ausdrücke der Form `let` x `=` e `in` e' im Ausdruck e' ein Vorkommen von x durch e ersetzt, wenn x nur einmal in e' vorkommt. Als Beispiel lässt sich folgende modifizierte `id`-Funktion betrachten.

```
id :: a -> a
id x = let y = x
       in y
```

Das `y` kommt nur einmal vor, also kann man es durch das x direkt ersetzen, sodass `id` wieder der Standardimplementierung

```
id :: a -> a
id x = x
```

entspricht.
Das *Let Inlining* könnte zum Beispiel nach der Guard Elimination in der Pipeline ergänzt werden, um die generierten `let`-Ausdrücke zu vereinfachen.

Ersetzung von Fallunterscheidungen
Da bereits Pattern Matching in `case`-Ausdrücke übersetzt wird, kann man um die Sprachkonstrukte weiter einzugrenzen, Fallunterscheidungen in vollständige `case`-Ausdrücke umformen. Die folgende Fallunterscheidung

`if` b `then` e_1 `else` e_2

wird dann zu

`case` b `of`
`True` $\rightarrow$ e_1
`False` $\rightarrow$ e_2

umgeformt.

Pattern Matching auf Zahlen übersetzen
In Haskell ist es möglich, auf `Integer` und `Int` zu Pattern Matchen. Als Beispiel kann man die naive Implementierung der Fibonacci-Zahlen betrachten.

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Mit der in Abschnitt 2.4 beschriebenen Transformation ist es nicht möglich, diese Funktion zu übersetzen, da die interne *Case Completion* dann unendlich viele Fälle hinzufügen müsste. Aber durch explizite Gleichheit lässt sich der Ausdruck trotzdem transformieren. Der generierte `case`-Ausdruck sieht dann wie folgt aus.

```
fib :: Integer -> Integer
fib x = case x == 0 of
          True  -> 1
          False -> case x == 1 of
                     True  -> 1
                     False -> fib (x-1) + fib (x-2)
```

Jedes Pattern Matching auf eine Zahl wird dabei zu einem `case`-Ausdruck transformiert.

Case Completion von `let`-Ausdrücken
In Haskell sind `case`-Ausdrücke der Form `let` $C\ p_1 \dots p_n$ `=` e `in` e' erlaubt, wobei C ein n-stelliger Konstruktor ist und $p_1 \dots p_n$ Pattern sind. Dies kann aber zu Fehlern beim Pattern Matching führen. Um also `let`-Ausdrücke dieser Form zu transformieren, benötigt man für den Konstruktor Selektorfunktionen für jeden Parameter. Dann ist

`let` $C\ p_1 \dots p_n = e$ `in` e'

zu

`let` $p_1 =$ $\texttt{select}_1\ e$
$\vdots$
$p_n =$ $\texttt{select}_n\ e$
`in` e'

äquivalent. Dabei ist $\texttt{select}_n$ die Selektorfunktion für den n-ten Parameter.

Anhang A

Anhang

A.1. Installation und Nutzung

Git-Struktur Die Implementierung, die in dieser Arbeit beschrieben wird, befindet sich in einem Git-Repository[1]
Das Git ist wie folgt strukturiert.

- src
  Enthält den Quelltext der Implementierung und die Tests für das cabal-Paket.

- LaTeX
  Enthält die LaTeX-Dateien dieser Bachelorarbeit.

- Examples
  Enthält einige Beispiele, die von Hand mit Hilfe des Algorithmus übersetzt wurden. Außerdem enthält der Ordner auch zwei Beispielhafte Queue-Imlementierungen zum Testen bestimmter Features.

Benötigte Software
Die Implementierung ist in Haskell geschrieben und benutzt Cabal, um die Abhängigkeiten von anderen Paketen zu verwalten. Um das Tool zu kompilieren muss Haskell[2] und Cabal[3] installiert sein. Das Tool wurde unter Windows mit Cabal Version 2.2.0.1 und GHC Version 8.4.3 kompiliert und getestet.

Benutzung
Nachdem die Executable mithilfe von cabal installiert wurde kann man mittels
`> CodeTransformation [Optionen..] < Dateipfade>` Das Tool benutzen. Dabei ist es Pflicht bei den Optionen mittels `-I` eine Input-Datei anzugeben. Mithilfe weiterer Flags können die Optionen für den Übersetzungsprozess eingestellt werden.

- -?, -h
  Diese Flags zeigen einem die Optionen an und es findet keine Übersetzung statt.

[1] `https://git.informatik.uni-kiel.de/stu204333/placc-thesis`
[2] `https://www.haskell.org/ghc/`
[3] `https://www.haskell.org/cabal/`

- -d
  Dieses Flag dient dem Anzeigen der Debug Informationen. Der Aktuelle Stand der Implementierung unterstützt dies jedoch nicht vollständig. Fehler im Übersetzungsprozess werden als Exception geworfen und auf der Konsole ausgegeben.

- -t
  Wenn dieses Flag gesetzt ist, wird die triviale Case Completion anstelle der Case Completion mit Hilfe des Algorithmus benutzt.

- -n
  Wenn dieses Flag gesetzt ist, dann werden die Optimierungen für doppeltes Pattern Matching ausgeschaltet.

- -o
  Wenn der Output in eine Datei geschrieben werden soll, dann muss dieses Flag gesetzt werden und ein Dateipfad für den Output angegeben werden.

Literaturverzeichnis

Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39677-2.

Edsger W. Dijkstra. Go to Statement Considered Harmful. In Manfred Broy and Ernst Denert, editors, *Software Pioneers: Contributions to Software Engineering*, pages 351–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-642-59412-0.

Simon P. Jones. The Haskell 98 Report: Formal Semantics of Pattern Matching.

Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction*, pages 258–270, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-47335-0.

Philip Wadler. Efficient Compilation of Pattern-Matching. *The Implementation of Functional Programming Languages*, 1987. URL `https://ci.nii.ac.jp/naid/10021860546/en/`.