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

Bachelorarbeit

Optimierung von Case-Ausdrücken in Curry-Programmen

Tom Hüser

September 2016

betreut von
Prof. Dr. Michael Hanus
M. Sc. Björn Peemöller

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Wattenbek, 26. September 2016

Tom Hüser

Inhaltsverzeichnis

1. Einleitung

Ein Compiler übersetzt beim Kompilieren eines Programms den geschriebenen Code so, dass auch eventuell vorhandener syntaktischer Zucker entfernt wird. Damit kann der Umfang des Codes in der kompilierten Form sehr viel größer werden, als in der geschriebenen Form. Dies ist unter anderem auch bei der funktional-logischen Programmiersprache Curry möglich. Hier kann das Problem spezifisch bei der Verwendung von Fallunterscheidungen in Form von Case-Ausdrücken auftreten.

Allerdings ist dies nicht bei jedem Case-Ausdruck der Fall, sondern eher bei Case-Ausdrücken, die mit Curry-Datentypen mit mindestens drei Konstruktoren arbeiten und nicht für jeden Konstruktor eine eigene Alternative haben. Je mehr Konstruktoren ein Datentyp hat, desto umfangreicher kann auch der Code werden. Hierbei wird aber durch den Compiler nicht neuer Code hinzugefügt, sondern bereits vorhandener Code mehrfach dupliziert. Dadurch ist nicht nur die Zeit des Kompilierens erhöht, sondern auch die Größe der kompilierten Datei ein Vielfaches der Quelldatei.

In dieser Arbeit soll ein Algorithmus implementiert werden, der Case-Ausdrücke in Curry-Programmen so optimieren kann, dass die Größe des kompilierten Codes so minimal wie möglich ansteigt. Als Grundlage wird dafür ein Pattern-Matching-Algorithmus von Simon L. Peyton Jones benutzt [5] und zum weiteren Verständnis ein Artikel von Geoff Barrett und Philip Wadler [2] herangezogen.

Hierzu werden zunächst in Kapitel 2 die Grundlagen erläutert, die für das Verständnis dieser Arbeit nötig sind. Dazu gehören die Programmiersprache Curry selbst, Substitution in Curry und AbstractCurry, welches eine Curry-Darstellungsform für Curry-Code selbst ist. In Kapitel 3 wird der oben erwähnte Algorithmus aus [5] erklärt und wie er in Curry implementiert werden kann. Anschließend wird in Kapitel 4 auf Optimierungen dieses Algorithmus eingegangen und unter anderem auch ein Lösungsansatz für das Problem der Code-Duplizierung gegeben. Abschließend wird der Algorithmus in Kapitel 5 mit implementierten Optimierungen evaluiert und ein Ausblick auf weitere Verbesserungen oder Veränderungen gegeben.

2. Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die zum Verstehen der Arbeit nötig sind. Dabei gibt es zunächst eine kurze Einführung in die Programmiersprache Curry, da in dieser Arbeit ein Algorithmus entwickelt wird, der selbst in Curry implementiert ist und Curry-Programme transformieren soll. Anschließend werden Substitutionen und Shadowing in Curry-Programmen erläutert, allerdings nicht mit mathematischen Grundlagen. Am Ende gibt es eine Einführung in AbstractCurry, da der Algorithmus auf der Ebene von AbstractCurry arbeiten wird.

2.1. Curry

Die Programmiersprache Curry ist eine deklarative Sprache, die die funktionale Programmierung mit der logischen Programmierung vereint. Sie verbindet also die wichtigsten Eigenschaften der funktionalen Programmierung, wie verschachtelte Ausdrücke und Lazy Evaluation, mit denen der logischen Programmierung, wie logische Variablen. Da Curry die Sprache Haskell abgesehen von der Überladung durch Typklassen erweitert, gibt es in der Syntax zwischen diesen beiden Sprachen große Ähnlichkeiten. In dieser Arbeit wird das Portland Aachen Kiel Curry System (PAKCS)[1] als Implementierung von Curry benutzt.

Im Folgenden gibt es einen kurzen Überblick über die Syntax und Semantik für Curry, basierend auf dem zum Zeitpunkt dieser Arbeit aktuellen Curry-Report [3] und dem Curry-Tutorial [1].

2.1.1. Datentypen

Mit dem Schlüsselwort `data` können neue Datentypen in Curry angegeben werden. Eine solche Deklaration hat die Form:

$$\texttt{data T = C}_1\ \tau_{11} \ldots \tau_{1n_1}\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{C}_k\ \tau_{k1} \ldots \tau_{kn_k}$$

[1] Version 1.14.1, verfügbar auf `https://www.informatik.uni-kiel.de/~pakcs/`

Damit ist ein neuer Datentyp mit dem Namen `T` und k Konstruktoren $\texttt{C}_1, \ldots, \texttt{C}_k$ deklariert. Jeder Konstruktor $\texttt{C}_i$ hat dabei den Typ

$$\tau_{i1} \texttt{ -> } \ldots \texttt{ -> } \tau_{in_i} \texttt{ -> T}$$

Hiermit lässt sich zum Beispiel ein Binärbaum für ganze Zahlen wie folgt definieren:

```
data IntTree = Leaf Int | Node IntTree Int IntTree
```

Ein solcher Baum besteht also entweder aus einem Blatt mit einer ganzen Zahl als Beschriftung oder aus einem Knoten mit Beschriftung und einem linken und rechten Teilbaum. Es gibt allerdings auch oft Fälle, wo Bäume benötigt werden, die nicht ganze Zahlen als Beschriftungen zulassen sondern einen anderen Datentyp. Damit nicht für jeden Baumtypen ein neuer Datentyp geschrieben werden muss, gibt es in Curry ebenfalls eine Datentyp-Deklaration mit Typvariablen von der Form:

$$\texttt{data T } \alpha_1 \ldots \alpha_m \texttt{ = C}_1 \ \tau_{11} \ldots \tau_{1n_1} \texttt{ | } \ldots \texttt{ | C}_k \ \tau_{k1} \ldots \tau_{kn_k}$$

Ein solcher Typ hat also m Typvariablen, und die Konstruktoren $\texttt{C}_i$ haben dann den Typ

$$\tau_{i1} \texttt{ -> } \ldots \texttt{ -> } \tau_{in_i} \texttt{ -> T } \alpha_1 \ldots \alpha_m$$

Jeder Typ τ_{ij} ist hier aus den Variablen $\alpha_1 \ldots \alpha_m$ und einem Typkonstruktor erstellt. Die Deklaration eines Datentyps für allgemeine Binärbäume kann damit analog zu oben wie folgt aussehen:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

Der Parameter `a` gibt den Typ der Baumbeschriftung an.

Mit dem allgemeinen Binärbaum kann der oben beschriebene Datentyp `IntTree` auch dargestellt werden durch:

```
type IntTree = Tree Int
```

Mit dem Schlüsselwort `type` wird dabei in Curry ein Typsynonym angegeben. In einem Programm können also nun `IntTree` oder `Tree Int` äquivalent benutzt werden, ohne dass etwas an der Typisierung des Programmes geändert wird. Typsynonyme dienen dazu Typdefinitionen leichter lesbar zu machen.

2.1.2. Funktionen

In Curry ist eine Funktion definiert durch eine optionale Signatur und eine Reihe von Regeln. Falls die Signatur weggelassen wird, wird der Typ der Funktion durch den Compiler inferiert. Allgemein ist eine Funktion mit dem Namen `f` mit Signatur und einer Regel wie folgt definiert:

```
f :: τ1 -> τ2 -> ... -> τk -> τ
f p1 p2 ... pk = e
```

Dabei sind $\tau_1, \ldots, \tau_k, \tau$ Typausdrücke und `p1 ... pk` Patterns vom jeweiligen Typ. Das `e` auf der rechten Seite ist der Funktionsrumpf vom Typ τ und kann beispielsweise eine Zahl, ein einfacher Ausdruck oder auch eine Funktionsanwendung sein. Es können mehrere Regeln angegeben werden, die sich dann in den Patterns unterscheiden können. Mittels Pattern-Matching wird dann die Regel gewählt, die auf den aktuellen Fall anwendbar ist.

Ein einfaches Beispiel für eine konkrete Funktion ist eine Funktion, die aus den oben definierten `Tree` den Wert des ganz linken Blatts zurück gibt. Der Typ der Funktion wird durch die Signatur deutlich:

```
getLeftmostValue :: Tree a -> a
```

Diese Funktion ist also für Bäume eines beliebigen Typs definiert. Um nun das ganz linke Blatt zu finden, muss rekursiv immer wieder in die tieferen Schichten des linken Teilbaumes geschaut werden. Sobald das Blatt gefunden ist, wird der Wert zurückgegeben. Damit sieht die Funktion wie folgt aus:

```
getLeftmostValue (Leaf x)      = x
getLeftmostValue (Node l _ _) = getLeftmostValue l
```

Diese Funktion benutzt also Pattern-Matching und hat bei einer Regel auf der rechten Seite einen Ausdruck des Typs `a` und in der anderen Regel einen rekursiven Funktionsaufruf. Die Patterns enthalten hier Konstruktoren mit Variablen. Es wird hier durch das Pattern-Matching die Regel gewählt, die zum angewendeten Fall passen. Dabei passen Variablen immer und Konstruktoren nur für den jeweiligen Fall. Variablen werden an den konkreten Wert gebunden, womit sie auf der rechten Seite der Regel benutzt werden können. Die Variable `_` (Unterstrich) ist dabei ein Spezialfall, da sie ebenfalls immer passt, aber der Wert nicht gebunden wird. Es ist daher ratsam den Unterstrich immer dann zu verwenden, wenn ein Wert auf der rechten Seite nicht benötigt wird, wie hier im Beispiel der Wert des Knotens und der rechte Teilbaum.

2.1.3. Lokale Deklarationen

Damit bei einem Programm nicht alle Hilfsdeklarationen von Variablen oder Funktionen global sichtbar sein müssen, gibt es in Curry die Möglichkeit der lokalen Deklarationen. Dies kann durch `let`-Ausdrücke oder `where` bei entsprechenden Regeln gemacht werden.

Ein `let`-Ausdruck hat die Form `let decls in exp` und kann in `decls` Funktionen oder Variablen einführen, die nur in dem Ausdruck `exp` und den rechten Seiten der Deklarationen in `decls` sichtbar sind. Wird der `let`-Ausdruck als die rechte Seite einer Regel benutzt, so sind in den lokalen Deklarationen auch Variablen aus den Patterns sichtbar. Ein Beispiel für eine einfache Funktion mit `let`-Ausdruck ist `local1`:

```
local1 x = let a = 3 * b
               b = 2 * x
           in a + 2
```

Der Ausdruck `a + 2` ist hier äquivalent zu `3 * (2 * x) + 2`. Sollte es komplexere Ausdrücke in einem Ausdruck geben oder ein Ausdruck mehrfach verwendet werden, dient es der Übersichtlichkeit des Codes, wenn diese Ausdrücke in einem `let` vorher eingeführt werden.

Alternativ zu einem `let`-Ausdruck kann auch ein `where` für eine Regel angegeben werden. In den meisten Fällen erfüllt dies dieselbe Funktion wie ein `let`-Ausdruck, kann aber je nach Aufgabe zu besserer oder auch schlechterer Lesbarkeit des Codes führen. Die folgende Funktion `local2` macht genau das gleiche wie `local1`, benutzt aber `where` statt `let`:

```
local2 x = a + 2
  where
    a = 3 + b
    b = 2 * x
```

In manchen Fällen ist es erwünscht, ganze Strukturen von der linken Seite einer Regel auf der rechten Seite wiederzuverwenden. Dies ist möglich durch die Verwendung von so genannten As-Patterns. Hierbei wird durch die Form `v@pat` die Struktur der Pattern `pat` durch die Variable `v` identifiziert. Ein Beispiel dafür ist die folgende Funktion:

```
dropFalse (False:ys)  = ys
dropFalse xs@(True:_) = xs
```

Die Struktur `(True:_)` kann also auf der rechten Seite verwendet werden, indem die Variable `xs` benutzt wird. Die Verwendung von As-Patterns ist vor allem dann

interessant, wenn man nur an der Struktur einer Pattern interessiert ist und nicht an dem Inhalt. Bei dem oberen Beispiel ist nur relevant, dass die Liste mit `True` beginnt und die Restliste ist unwichtig. In solchen Fällen ist die Benutzung eines As-Patterns effizienter als zum Beispiel die Benutzung einer Hilfsfunktion, um eine Struktur zu überprüfen.

2.1.4. Nichtdeterminismus

Während bei Haskell bei überlappenden Patterns immer die der Reihenfolge nach zuerst passende Regel einer Funktion benutzt wird, wird ein solcher Fall in Curry nichtdeterministisch behandelt. Das bedeutet, dass alle passenden Regeln angewandt werden, egal in welcher Reihenfolge sie stehen. Ein Beispiel dafür ist die Funktion `ndet`:

```
ndet []      = 0
ndet _       = 42
```

In Haskell würde diese Funktion `0` zurückgeben, wenn sie mit einer leeren Liste aufgerufen wird und in allen anderen Fällen - hier also nur bei nicht-leeren Listen - den Wert `42`. Bei Curry wird bei einer nicht-leeren Liste ebenfalls nur `42` zurückgegeben. Da für eine leere Liste aber beide Regeln anwendbar sind, werden in diesem Fall beide Werte zurückgegeben. Bei diesem Beispiel kann, um Nichtdeterminismus zu vermeiden, der Unterstrich durch `(_ : _)` ersetzt werden, was auf beliebige nicht-leere Listen matcht.

2.1.5. Case-Ausdrücke

Da in dieser Arbeit Case-Ausdrücke in Curry-Programmen transformiert werden sollen, sind diese natürlich auch in Curry vorhanden. Die einfachste Form eines Case-Ausdrucks wird als rigider Case-Ausdruck bezeichnet und ist von der Form:

```
case e of
  p1 -> e1
  ⋮
  pn -> en
```

Dabei sind $e, e_1, \ldots, e_n$ Ausdrücke und die Patterns $p_1, \ldots, p_n$ Variablen- oder Konstruktor-Pattern. Bei einem solchen Case-Ausdruck wird so lange der Wert von e überprüft, bis er mit einem Pattern p_i übereinstimmt. Der gesamte Case-Ausdruck kann dann durch den Ausdruck e_i ersetzt werden. Sollte keine der Patterns mit e übereinstimmen, so scheitert die Berechnung. Die Berechnung kommt auch zu einem Ende, wenn e zu

einer freien Variablen ausgewertet wird, falls also ein p_i eine freie Variable ist, wie in bei `ndet` der Unterstrich. Damit lassen sich Case-Ausdrücke gut benutzen, um einen Standardfall für Funktionen zu schreiben.

Rigide Case-Ausdrücke können auch dazu dienen, um Nichtdeterminismus zu vermeiden. Es ist in manchen Fällen sinnvoll überlappende Regeln zu nutzen, ohne dass diese nichtdeterministisch ausgewertet werden. Dies ist der Fall, wenn in `ndet` zum Beispiel nicht auf Listen gematcht wird, sondern auf einen anderen Datentypen mit mehr als zwei Konstruktoren. Die Funktion soll den Rückgabewert `0` für einen dieser Typen haben und für alle anderen `42`. Anstatt eine Regel für jeden dieser Konstruktoren zu schreiben, kann hierfür ein Case-Ausdruck benutzt werden, da dieser in Curry deterministitsch ist und die Patterns wie bei Haskell der Reihenfolge nach getestet werden. Die Funktion `ndet` kann also deterministitsch wie folgt geschrieben werden:

```
det x = case x of
          [] -> 0
          _  -> 42
```

Für eine leere Liste wird also nur `0` zurückgegeben und für alle andern Fälle `42`.

Falls bei einem Case-Ausdruck der Nichtdeterminismus dennoch erwünscht ist, so können flexible Case-Ausdrücke verwendet werden. Bei diesen wird nicht nur eine Alternative durch Pattern-Matching der Reihenfolge nach gewählt. Sie folgen dem nichtdeterministischen Pattern-Matching, wie es auch bei Funktionen der Fall ist. Ein solcher flexibler Case-Ausdruck hat die Form:

```
fcase e of
  p1 -> e1
  ⋮
  pn -> en
```

Hier sind wieder $e, e_1, \ldots, e_n$ Ausdrücke und die Patterns $p_1, \ldots, p_n$ Variablen- oder Konstruktor-Pattern. Der Ausdruck schreibt sich also genau so wie ein rigider Case-Ausdruck, allerdings mit dem Schlüsselwort `fcase` statt `case`. In der Funktionalität entspricht dieser flexible Case-Ausdruck eher dem folgenden Ausdruck:

```
let f p1 = e1
  ⋮
    f pn = en
 in f e
```

Hierbei ist `f` ein frisches Hilfssymbol für eine Funktion. Bei dieser Auswertung können auf Grund des flexiblen Pattern-Matchings mehrere Alternativen gewählt werden. Wenn in der Funktion `det` mit dem Case-Ausdruck das `case` durch ein `fcase` ersetzt wird, verhält sie sich wieder so wie die Funktion `ndet` in Abschnitt 2.1.4.

Sowohl flexible als auch rigide Case-Ausdrücke können sogenannte Guards enthalten. Ein solches Guard hat den Typ `Bool` und stellt eine Bedingung für einen Fall des Case-Ausdrucks dar. Passt der Wert von `e` zu einem Pattern und hat dieses Pattern ein Guard, so wird dies überprüft. Falls das Guard erfüllt wird, wird der Case-Ausdruck weiter zu dem dazugehörigen Ausdruck ausgewertet. Sollten alle Guards dieses Falls zu `False` ausgewertet werden, wird der nächste Fall betrachtet. Es wird also so gehandhabt, als ob das erste Pattern gar nicht gepasst hätte. Ein Guard wird dabei durch `|` eingeführt und ist äquivalent zu einem passenden `if-then-else` Ausdruck. Ein solcher Case-Ausdruck hat die Form:

```
case e of
  p1 | b11 -> e11
     ⋮
     | b1m1 -> e1m1
  ⋮
  pn | bn1 -> en1
     ⋮
     | bnmn -> enmn
```

Dabei sind b_{ij} die Guards. Es kann beliebig viele Guards für einen Fall geben und es können manche Fälle auch gar keine Guards haben.

Bei rigiden Case-Ausdrücken kann zwischen normalisierten und nicht-normalisierten Ausdrücken unterschieden werden. Jeder rigide Case-Ausdruck kann normalisiert werden. Ein normalisierter Case-Ausdruck hat allerdings keine Variablen als Patterns, sondern nur die entsprechenden Konstruktoren des Typs von `e` angewandt auf Variablen. Das bedeutet: Es wird eine Variable gegen verschiedene Konstruktoren abgeglichen, jeder Fall muss abgedeckt sein und die Patterns dieser Konstruktoren müssen alle Variablen sein. Eine normalisierte Form von dem Case-Ausdruck für `det` ist damit beispielsweise:

```
det x = case x of
          []    -> 0
          (_:_) -> 42
```

Ein Curry-Compiler normalisiert solche Case-Ausdrücke beim Kompilieren, um Pattern-Matching zu ermöglichen.

2.2. Substitution

In manchen Fällen ist es wünschenswert in einem Ausdruck eine Variable durch einen anderen Ausdruck zu substituieren. Dabei muss jedes Vorkommen dieser Variablen auch in verschachtelten Unterausdrücken ersetzt werden. Vor allem bei Curry ist es wichtig, dass die Substitution hier korrekt abläuft, da in lokalen Deklarationen von Unterausdrücken auch Variablen benutzt werden dürfen, die zuvor schon gebunden wurden. Solch eine Benutzung wird als „Shadowing“ bezeichnet. Ein einfaches Beispiel für eine Funktion mit Shadowing ist die folgende Funktion `shadow :: [Int] -> Int`:

```
shadow x = case x of
             []    -> let x = 42
                      in x
             [x]   -> x - 2
             (_:_) -> sum x
```

Die Variable `x` wird in dem Pattern an den übergebenen Parameter gebunden. In den beiden ersten Fällen des Case-Ausdrucks tritt dann Shadowing auf. Zum einen wird `x` in der `let`-Deklaration neu an den Wert `42` gebunden und im anderen Fall in dem Pattern an das Element der einelementigen Liste. Beide Fälle funktionieren und geben auch den korrekten Wert zurück. Auf den rechten Seiten hat `x` dann immer den neu gebundenen Wert statt des ursprünglich gebundenen. Im letzten Fall ist der Wert von `x` noch der alte Wert, sodass mit der Funktion `sum` die Summe aller Zahlen in der Liste zurückgegeben wird. Sofern der alte Wert noch benötigt wird, sollte Shadowing vermieden werden.

Da Shadowing aber grundsätzlich möglich ist, muss bei der Substitution darauf geachtet werden, dass mit Shadowing neu definierte Variablen nicht substituiert werden. Sollte im oberen Beispiel die ursprüngliche Variable `x` durch die neue Variable `y` substituiert werden, sieht die Funktion wie folgt aus:

```
shadow y = case y of
             []    -> let x = 42
                      in x
             [x]   -> x - 2
             (_:_) -> sum y
```

Es sind nur die Vorkommen der Variablen `x` in der ersten und letzten Zeile ersetzt worden. Die neu gesetzten `x` wurden ignoriert, so dass in diesem Fall kein Shadowing mehr vorliegt und die Funktionalität nach wie vor dieselbe ist. Würde die Variable auch in

dem `in` Teil des `let` substituiert oder auf der rechten Seite bei der einelementigen Liste, würde dies zu einem Fehler führen, da hier ein Integer erwartet wird und keine Liste. Zu beachten ist, dass hier kein Shadowing mehr vorliegt, obwohl es zweimal die Variable `x` mit unterschiedlicher Bindung gibt. Das liegt daran, dass sie in unterschiedlichen Sichtbarkeitsbereichen sind und nur auf der rechten Seite der jeweiligen Alternativen zu sehen sind.

2.3. AbstractCurry-Grundlagen

Hier werden die Grundlagen für AbstractCurry erklärt, die im Verlauf der Arbeit genutzt werden. Patterns in einem AbstractCurry-Programm werden durch den Datentyp `CPattern` repräsentiert. Die hier relevanten `CPattern` sind die folgenden:

```
CPVar  :: CVarIName -> CPattern
CPComb :: QName     -> [CPattern] -> [CPattern]
CPLit  :: CLiteral  -> CPattern
```

Dabei ist `CVarIName` ein Synonym für `(Int, String)` und besteht aus einem Variablenindex und Variablennamen. `QName` ist ein qualifizierter Name und ist ein Typsynonym für `(MName, String)`, wobei `MName` ein Typsynonym für `String` ist und den Modulnamen darstellt und die zweite Komponente der Name ist. Somit hat eine Variable also einen Namen und ein Konstruktor einen qualifizierten Namen und Patterns. Ein `CLiteral` kann ein Integer, Float oder Buchstabe sein.

Eine Konstruktordeklaration eines Curry-Programms wird durch den Datentypen `CConsDecl` und den Konstruktor

```
CCons :: QName -> CVisibility -> [CTypeExpr] -> CConsDecl
```

dargestellt. Die `[CTypeExpr]` sind die Typen, die der Konstruktor benötigt, siehe dazu Abschnitt 2.1.1. Die `CVisibility` bestimmt die Sichtbarkeit des Konstruktors bezüglich des Exports. Ist sie `Public`, so wird der Konstruktor bei dem Modul mitexportiert und kann damit von anderen Modulen benutzt werden. Wenn sie `Private` ist, wird er nicht exportiert.

Ausdrücke werden durch den Datentypen `CExpr` repräsentiert und, obwohl es mehrere gibt, ist der hier Wichtigste

```
CCase :: CCaseType -> CExpr -> [(CPattern, CRhs)] -> CExpr.
```

Dieser stellt Case-Ausdrücke in AbstractCurry dar. `CCaseType` kann `CRigid` oder `CFlex` sein, wobei in dieser Arbeit nur der erste betrachtet wird, da der zweite für

flexible Case-Ausdrücke ist. Der zweite Parameter ist der Ausdruck, der durch den Case-Ausdruck überprüft wird und die Liste von `(CPattern, CRhs)` werden in dieser Arbeit als Klauseln bezeichnet.

Eine `CRhs` ist eine rechte Seite eines Case-Ausdrucks oder einer Regel. Hier gibt es nur die zwei Konstruktoren für unbedingte und bedingte rechte Seiten:

```
CSimpleRhs  :: CExpr -> [CLocalDecl] -> CRhs
CGuardedRhs :: [(CExpr,CExpr)] -> [CLocalDecl] -> CRhs
```

Eine `CSimpleRhs` hat also einen Ausdruck und eine Liste lokaler Deklarationen und eine `CGuardedRhs` hat eine Liste von Guards mit entsprechenden rechten Seiten und eine Liste lokaler Deklarationen.

Der Datentyp `CLocalDecl` steht für lokale Deklarationen, wie sie in `let` und `where` vorkommen. Er hat die drei Konstruktoren

```
CLocalFunc :: CFuncDecl -> CLocalDecl
CLocalVars :: [CVarIName] -> CLocalDecl
CLocalPat  :: CPattern -> CRhs
```

für lokale Funktionen, für freie logische Variablen und für lokale Patterns. Mit dem letzten Konstruktor können also lokale Deklarationen erstellt werden, wie sie in Abschnitt 2.1.3 benutzt wurden. Die `CRhs` ist die rechte Seite, die dem `CPattern` zugeordnet wird.

Ein Curry Programm selbst kann auch in AbstractCurry dargestellt werden, und zwar als Datentyp

```
CurryProg :: MName -> [MName] -> [CTypeDecl]
          -> [CFuncDecl] -> [COpDecl] -> CurryProg
```

Der Datentyp hat damit die Form `(CurryProg mname imports tdecls funcs opdecls)`, wobei `mname` der Name des Programms oder Moduls ist. `imports` bezeichnet die Namensliste der importierten Module, `tdecls` Typdeklarationen, `funcs` Funktionsdeklarationen und `opdelcs` Prioritätsdeklarationen für Operatoren.

Die im vorherigen Abschnitt erwähnten Gleichungen sind Paare von Patternlisten und Ausdrücken, also `([CPattern], CExpr)`, und repräsentieren die Fälle eines Case-Ausdrucks für die Umsetzung des Algorithmus.

3. Implementierung

In diesem Abschnitt wird das Konzept hinter dem Pattern-Matching-Algorithmus von Simon L. Peyton Jones [5] kurz zusammengefasst und erklärt. Der Algorithmus, der für das Kompilieren von Sprachen, die mit Pattern-Matching arbeiten, zuständig ist, bietet die Grundlage für den Algorithmus dieser Arbeit. Hier sollen Case-Ausdrücke in Curry-Programmen normalisiert werden. Anschließend wird auf die Umsetzung in Curry und auf die Anwendung des Algorithmus auf ein Curry-Programm eingegangen.

3.1. Der Algorithmus

Das Ziel ist es, einen Pattern-Matching-Compiler-Algorithmus zu erhalten, der mittels einer Funktion match die Case-Ausdrücke einer Funktion in äquivalente, normalisierte Case-Ausdrücke umwandelt. Dazu müssen vor Anwendung des Algorithmus diese Case-Ausdrücke so umgeformt werden, dass keine Guards oder `where`-Deklarationen mehr vorkommen. Wie dies umgesetzt ist, wird in Abschnitt 3.2.2 genauer beschrieben. Die match-Funktion erhält als Parameter eine Liste von freien Variablen, eine Liste von Patterns mit Ausdrücken, hier als Gleichungen bezeichnet, und einen Default-Ausdruck. Die Gleichungen werden in der Implementierung aus den Alternativen eines Case-Ausdrucks gewonnen. Es werden dabei so viele frische Variablen an match übergeben, wie die Arität der Patterns ist, womit es eine Variable pro Stelle gibt. Das bedeutet ein Aufruf von match hat die Gestalt:

```
match [u1,u2 ..., un]
      [([p11, p12,..., p1n], e1),
          ⋮
       [pm1, pm2,..., pmn], em)]
      def
```

Dabei bezeichnen hier u_j die frischen Variablen, p_{ij} die Patterns, e_i die dazugehörigen Ausdrücke und `def` den gewählten Default-Ausdruck. Je nach Art der Patterns müssen unterschiedliche Regeln angewandt werden, die im Folgenden kurz vorgestellt werden.

Der Übersichtlichkeit halber werden Curry-ähnliche Schreibweisen benutzt wie zum Beispiel `(u:us)` für `[`u_1`,`$u_2 \ldots$`, `u_n`]`, wobei `u` für u_1 steht und `us` für die Restliste `[`$u_2 \ldots$`, `u_n`]`.

3.1.1. Die Variablen-Regel

Sollten alle Patternlisten als ersten Parameter eine Variable besitzen, so wird die Variablen-Regel angewandt. Dabei wird diese erste Variable durch die erste Variable aus der Liste freier Variablen in dem zum Pattern gehörenden Ausdruck substituiert. Gleichzeitig wird die erste freie Variable aus der Liste entfernt und ebenso der erste Parameter aus der Patternliste. Damit gibt es für den nächsten match-Aufruf noch immer so viele freie Variablen wie es Parameter in den Patterns gibt. Die Variablen-Regel ist also ein Aufruf der Form:

```
match (u:us)
      [ ( (v1 : ps1), e1),
              ⋮
        ( (vm : psm), em)]
      def
```

Die v_i sind hier also Patterns, die alle Variablen sind. Dieser Aufruf wird umgewandelt zu einem Aufruf der Form:

```
match us
      [ ( ps1, e1[u/v1]),
              ⋮
        ( psm, em[u/vm])]
      def
```

Dabei steht $e_i[u/v_i]$ für den Ausdruck e_i mit v_i substituiert durch `u`. Ein Beispiel hierzu ist der folgende match-Aufruf:

```
match [u1, u2]
      [ ( [x, []],   (A x) ),
        ( [y, x:xs], (B y x xs) )]
      failed
```

`A` und `B` sind dabei nicht näher spezifizierte Konstruktoren desselben Typs. Dieser Aufruf von match wird dann wie folgt ungewandelt:

```
match [u2]
      [ ( [[]],   (A u1) ),
        ( [x:xs], (B u1 x xs) )]
      failed
```

Diese Methode funktioniert also auch dann, wenn die Gleichungen mit einer anderen Variablen beginnen.

3.1.2. Die Konstruktor-Regel

Besitzen alle Patterns als ersten Parameter ein Konstruktorpattern, so kommt die Konstruktor-Regel zum Einsatz. Hierbei wird der match-Aufruf durch einen äquivalenten Case-Ausdruck ersetzt, bei dem alle Ausdrücke zu dem jeweils passenden Konstruktor zugeordnet werden. Das bedeutet, dass Gleichungen, die mit demselben Konstruktor beginnen, gruppiert werden. Allerdings müssen Gleichungen, die mit demselben Konstruktor anfangen, in den Regeln nicht unbedingt hintereinander vorkommen, womit durch das Gruppieren die Reihenfolge der Regeln verändet wird. Allerdings kann man immer ohne Probleme Gleichungen, die mit einem anderen Konstruktor beginnen, austauschen, sofern die Reihenfolge von gleichen Konstruktoren noch dieselbe ist, womit auch nach dem Umtauschen noch ein äquivalenter Ausdruck vorliegt [5]. Die rechten Seiten des Case-Ausdrucks sind weitere match-Aufrufe mit so vielen zusätzlichen freien Variablen, wie der entsprechende Konstruktor Argumente hat. Diese neuen freien Variablen werden dann gegen die Subpatterns der ursprünglichen Patterns abgeglichen und sind gleichzeitig die Parameter der Konstruktoren auf der linken Seite des Case-Ausdrucks. In dem Case-Ausdruck selbst wird die erste freie Variable überprüft und somit Pattern-Matching auf das erste Pattern angewandt.

Falls bei einem match-Aufruf nun alle Gleichungen mit einem Konstruktor beginnen und falls der Typ dieser Konstruktoren die Konstruktoren $C_1, \ldots, C_k$ hat, so können die Gleichungen in die Gruppen $qs_1, \ldots, qs_k$ eingeteilt werden, wobei jede Gleichung in qs_i mit dem Konstruktor C_i beginnt. Sollte es keine Gleichungen für einen Konstruktor geben, so bleibt diese Gruppe leer. Die einzelnen qs_i haben dann die folgende Gestalt:

```
[ ( ((Ci ps'i,1):psi,1), ei,1 )
    ⋮
  ( ((Ci ps'i,mi):psi,mi), ei,mi ) ]
```

Die $ps'_{i,j}$ sind die Patterns des Konstruktors C_i, hierbei steht also `(C ps)` als Abkürzung für `(C` $p_1 \ldots$ p_r`)` bei einem r-stelligen Konstruktor `c`. Ein match-Aufruf, bei dem die Konstruktor-Regel benutzt wird, hat damit die Form:

```
match (u:us) (qs_1 ++ ... ++ qs_k) def
```

Dabei bezeichnet `++` die Funktion `append`, also das Aneinanderhängen von Listen. Der obere Aufruf von `match` wird dann in den folgenden Case-Ausdruck umgewandelt:

```
case u of
  C_1 us'_1 -> match (us'_1 ++ us) qs'_1 def
  ⋮
  C_k us'_k -> match (us'_k ++ us) qs'_k def
```

wobei die `qs'`$_i$ nun die Patterns des Konstruktors zur Patternliste hinzubekommen und somit die folgende Form haben:

```
[ ( (ps'_{i,1} ++ ps_{i,1}), e_{i,1} )
    ⋮
  ( (ps'_{i,m_i} ++ ps_{i,m_i}), e_{i,m_i} ) ]
```

Die `us'`$_i$ sind Variablenlisten, die je eine Variable für jedes Feld des Konstruktors `C`$_i$ enthalten, bei einem r-stelligen Konstruktor also auch r Variablen. Da Konstruktoren eines Typs nicht alle die gleiche Stelligkeit besitzen müssen, können sich die Längen der einzelnen `us'`$_i$ also voneinander unterscheiden.

3.1.3. Die Leer-Regel

Nach mehrfacher Anwendung der oberen beiden Regeln wird es irgendwann einen match-Aufruf geben, bei dem die Variablenliste leer ist. In einem solchen Fall wird die Leer-Regel angewandt. Die Form eines solchen Aufrufs ist:

```
match []
      [ ( [], e_1),
            ⋮
        ( [], e_m)]
      def
```

Dieser Ausdruck wird zu `e`$_1$ ausgewertet, sofern $m > 0$ ist. In diesem Fall ist die Liste der Gleichungen nicht leer. Sie kann aber auch leer sein, sofern die Konstruktor-Regel angewandt wurde und es für einen Konstruktor des Typs keine Gleichungen gab. In diesem Fall ist $m = 0$ und der obere match-Aufruf wird zu dem Default-Ausdruck `def` ausgewertet.

3.1.4. Die Mischungs-Regel

Es kann aber auch sein, dass unterschiedliche Patterns einer Funktion nicht immer an der gleichen Stelle nur jeweils Konstruktoren oder Variablen enthalten, sondern diese durchmischt sind. Damit kann also weder die Variablen-Regel, noch die Konstruktor-Regel direkt angewandt werden. Ein einfaches Beispiel ist erneut die Funktion `det` aus Abschnitt 2.1.5. Der match-Aufruf dafür hat die Form:

```
match [u]
      [ ( [ [] ], 0),
        ( [ _  ], 42)]
      failed
```

Hier steht als Pattern also einmal der Konstruktor `[]` und an derselben Stelle in der anderen Gleichung die Variable `_`. Als Default Ausdruck ist `failed` übergeben, da hier nicht erfolgreiches Pattern-Matching zu einem Fehlschlag führt. Da keine der beiden Regeln angewandt werden kann, muss der Ausdruck umgeschrieben werden:

```
match [u]
      [ ( [ [] ], 0)]
      ( match [u]
              [( [ _  ], 42)]
              failed)
```

Hier wird der Default-Ausdruck des ersten match-Aufrufs durch einen zweiten match-Aufruf ersetzt, dessen Default-Ausdruck der alte – also `failed` – ist. Die Gleichungen wurden auf die beiden Aufrufe so verteilt, dass der erste nur die Gleichung mit Konstruktor enthält und der zweite die mit der Variablen. Die Funktionalität dieser Aufteilung ist äquivalent zu der ohne Trennung. Wird `u` zu `[]` ausgewertet, wird `0` zurückgegeben, ansonsten der Default-Ausdruck, der wegen der einstelligen Pattern mit nur einer Variablen immer zu `42` ausgewertet wird. Sollte dieser Fall nicht eintreten, was hier nicht möglich ist, würde `failed` zurückgegeben.

Die Liste der Gleichungen wird also auf mehrere match-Aufrufe aufgeteilt und zwar so, dass jeder Aufruf nur Gleichungen hat, die mit Variablen oder mit Konstruktoren beginnen. Damit können abwechselnd die Variablen-Regel und Konstruktor-Regel angewandt werden. Würde es in dem Beispiel oben noch eine dritte Gleichung geben, die mit einem Konstruktor beginnt, so gäbe es auch einen dritten Aufruf von match, dessen Ergebnis der Default-Ausdruck für den zweiten match-Aufruf ist.

Betrachten wir den folgenden allgemeinen Aufruf von match:

```
match us qs def
```

Angenommen die Liste der Gleichungen `qs` ist so, dass die Mischungs-Regel angewandt werden muss. Dann kann `qs` so unterteilt werden, dass

```
qs = qs1 ++ ... ++ qsk
```

ist. Dabei sollte die Unterteilung so sein, dass jede Liste $\texttt{qs}_i$ nur Gleichungen enthält, die entweder nur mit Variablen oder nur mit Konstruktoren beginnen. Jede Liste kann auch mehr als eine Gleichung erhalten, allerdings muss das Aneinanderhängen der Listen die Ursprungsliste `qs` ergeben, mit allen Gleichungen in derselben Reihenfolge wie vorher. Ist dies gegeben, so kann der Aufruf von `match` zu dem folgenden Aufruf reduziert werden:

```
match us qs1 (match us qs2 ( ... (match us qsk def) ...))
```

Die Gleichungen werden also so aufgeteilt, dass entweder die Variablen-Regel oder die Konstruktor-Regel angewandt werden kann.

3.2. Umsetzung in Curry

Der oben beschriebene Algorithmus aus dem Quellmaterial [5] kann auch Pattern-Matching für Funktionen machen. Hierbei ergeben sich die Gleichungen nicht aus Case-Ausdrücken sondern aus den Regeln der Funktion selbst. Da das Pattern-Matching in Curry aber unter anderem wegen des Nichtdeterminismus anders erfolgt, wird der Algorithmus hier nur verwendet, um bereits vorhandene Case-Ausdrücke zu transformieren. Das bedeutet, dass Funktionen, die keine Case-Ausdrücke enthalten, unverändert bleiben. Daher wird in diesem Abschnitt die Umsetzung des Algorithmus in Curry diskutiert und im nächsten Abschnitt wie der Algorithmus auf ein Programm angewandt wird. Die Umsetzung des Algorithmus an sich arbeitet auf der Ebene von AbstractCurry. Dafür wird ein Curry-Programm zunächst in ein AbstractCurry-Programm übertragen, auf das der Algorithmus angewandt wird. Anschließend kann man es wieder zurück in ein reguläres Curry-Programm übersetzen, um z.B. Änderungen zum Ursprungsprogramm zu betrachten. Die Funktionalität des Programmes ist aber nach wie vor dieselbe.

3.2.1. Die Funktion `match`

Da im späteren Verlauf des Algorithmus Variablen substituiert und neue frische Variablen generiert werden müssen, werden alle `match`-Aufrufe in eine State-Monade

verpackt. Diese sieht wie folgt aus und benutzt einen Zustand, wie er in der Bibliothek `State.curry` implementiert ist: `type PMM a = State SipeyState a`. `SipeyState` ist dabei ein Record der Form:

```
type TypeList = [(QName, [(QName, Int)])]

data SipeyState = SipeyState
  { freshIdx           :: Int
  , constructorTypes   :: TypeList
  , optRepeatedMatches :: Bool
  , optCodeDuplication :: Int
  }
```

Ein Record erlaubt es in Curry einfacher auf bestimmte Elemente zuzugreifen, als wenn man ein Tupel benutzt. Vor den `::` stehen die Selektoren, mit denen man auf die Elemente zugreifen kann, und dahinter der Typ dieses Elements. Das heißt der Zustand führt hier einen Zähler `freshIdx` und eine `TypeList` mit, die ein selbst deklarierter Datentyp ist und als Liste von Konstruktorentypen dient. Dabei ist das erste `QName` der Konstruktor und die darauffolgende Liste enthält alle Konstruktoren dieses Typs als `QName` gepaart mit der Arität des jeweiligen Konstruktors. Wie diese Liste generiert wird, wird später noch genauer erklärt. Die beiden letzten Selektoren sind Flags, um einfacheres Testen mit später eingeführten Optimierungen durchzuführen.

Frische Variablen werden durch die beiden Funktionen

```
freshVar :: PMM Int
makeVar  :: PMM CVarIName
```

eingeführt. In `freshVar` wird der aktuelle Wert des Zählers `freshIdx` zurückgegeben und dann erhöht, womit eine neue frische Variable mit anderem Zähler generiert werden kann.

In `makeVar` wird dann die Variable zurückgegeben, wobei der Index der Variablen 1 ist. Neu generierte Variablen erhalten den Namen `pm_ui`, wobei i der Zählerstand zum Zeitpunkt der Generierung ist. Der Präfix `"pm_u"` wurde so gewählt, damit es möglichst unwahrscheinlich ist, dass die neue Variable mit einer bereits im Code vorhandenen verwechselt werden kann.

Die Funktion `match` an sich hat dann die Gestalt wie in Listing 3.1. Der erste Parameter ist die Liste freier Variablen, der zweite sind die Patterns mit den entsprechenden Ausdrücken auf der rechten Seite, also die Gleichungen, und der dritte Parameter ist

```
match :: [CVarIName] -> [([CPattern], CExpr)] -> CExpr -> PMM CExpr
match []       []     def = returnS def
match []       (q:_) _   = returnS (snd q)
match us@(u:_) qs     def
| any isAs  qs = match us (map (removeAs u) qs) def
| all isVar qs = matchVar us qs def
| all isCon qs = matchCon us qs def
| all isLit qs = matchLit us qs def
| otherwise    = ... -- Mischungs-Regel
```

Listing 3.1.: Teilimplementierung der Funtkion match

für den Default-Ausdruck. Der Rückgabewert ist `PMM CExpr`, damit der Zustand weiter verwendet werden kann. Sollte es bei einem Aufruf von match keine Gleichungen und freie Variablen mehr geben, so gibt match den Default-Ausdruck zurück. Sind nur keine Variablen mehr vorhanden aber immer noch Gleichungen, gibt match den ersten Ausdruck in der ersten Gleichung zurück. Hiermit ist also die Leer-Regel abgedeckt. Sofern Variablen und Gleichungen noch vorhanden sind, muss die Form der Patterns unterschieden werden. Zunächst wird dazu überprüft, ob irgendeines der ersten Patterns ein As-Pattern ist. Da diese gleichzeitig eine Variable binden und ein Pattern sind, müssen sie so umgeändert werden, dass nur das daran gebundene Pattern an dieser Stelle steht. Dabei muss jedes Vorkommen der Variablen auf der rechten Seite durch einen äquivalenten Ausdruck ersetzt werden. Ein match-Aufruf mit einem As-Pattern kann zum Beispiel wie folgt aussehen:

```
match [u]
      [ ( [ [] ],       []),
        ( [ xs@(_:_) ], xs)]
      failed
```

Hier sollte eigentlich die Konstruktor-Regel angewandt werden, aber wegen des As-Patterns ist dies nicht so einfach möglich. Dieser match-Aufruf wird dann zu dem folgenden Aufruf umgewandelt:

```
match [u]
      [ ( [ [] ],    []),
        ( [ (_:_) ], u )]
      failed
```

Da für dieses Pattern die Variable `u` vorgesehen ist, kann die durch das As-Pattern gebundene Variable `xs` auf der rechten Seite also durch `u` substituiert werden. Auf der linken Seite wird das As-Pattern nur durch das entsprechende Pattern ersetzt. Die Funktion `removeAs` tut genau dies für eine Gleichung, die mit einem As-Pattern beginnt, alle anderen werden unverändert gelassen.

Anschließend wird überprüft, ob alle Patterns der Gleichungen mit einer Variablen, einem Konstruktor oder einem Literal anfangen. Sollte einer der ersten beiden Fälle eintreten, wird jeweils die Variablen- oder die Konstruktor-Regel angewandt, wobei diese Regeln in `matchVar` beziehungsweise `matchCon` umgesetzt sind. Da im Quellmaterial [5] nicht explizit auf Literale eingegangen wird und eine gesonderte Behandlung davon in Curry aber nötig ist, wird hier zusätzlich noch eine Literal-Regel eingeführt, die im Rahmen dieser Arbeit erarbeitet wurde. Diese ist in `matchLit` implementiert und wird im dritten Fall benutzt. Die Überprüfung der Patterns erfolgt über die Funktionen `isVar`, `isCon` und `isLit`, die auf eine Gleichung angewandt `True` zurückgeben, wenn die Patterns jeweils mit einer Variablen, einem Konstruktor oder einem Literal anfangen, ansonsten `False`.

Tritt keiner der oben genannten Fälle auf, so muss die Mischungs-Regel angewandt werden. Hier wird die Liste der Gleichungen `qs` in zwei Teile unterteilt, wobei in `qs1` jeweils das erste Pattern der Gleichungen entweder nur Variablen oder Konstruktoren oder Literale enthält und `qs2` alle diese in beliebiger Reihenfolge enthalten kann. Die Aufteilung behält die ursprüngliche Reihenfolge der Gleichungen und trennt nur die ersten Gleichungen desselben Typs vom Rest der Liste, bis eine Gleichung eines anderen Typs in der Liste `qs` auftaucht. Anschließend wird `match` zwei mal mit `qs1` oder `qs2` als Gleichungen aufgerufen, wobei der Default-Ausdruck des ersten Aufrufs der Rückgabewert des zweiten Aufrufs werden muss. Durch den rekurisven Aufruf von `match` mit `qs2` wird hier also erneut geprüft, ob die Mischungs-Regel angewandt werden muss oder nicht, womit die Unterteilung der ursprünglichen Liste von Gleichungen `qs` wie in Abschnitt 3.1.4 erfolgt. Hiermit ist die Mischungs-Regel abgedeckt.

Die Variablen-Regel ist in der Funktion

```
matchVar :: [CVarIName] -> [([CPattern], CExpr)]
         -> CExpr -> PMM CExpr
```

umgesetzt und wird nur dann angewandt, wenn alle ersten Patterns der Gleichungen eine Variable sind. Diese ersten Patterns werden für den nächsten `match`-Aufruf entfernt und die dazugehörige Variable in dem Ausdruck der Gleichung durch die erste freie Variable aus der Variablenliste substituiert.

Die Konstruktor-Regel ist in den zwei Funktionen

```
matchCon :: [CVarIName] -> [([CPattern], CExpr)]
         -> CExpr -> PMM CExpr
```

und

```
matchClause :: QName -> Int -> [CVarIName] -> [([CPattern], CExpr)]
            -> CExpr -> PMM (CPattern, CRhs)
```

implementiert. Da alle Patterns mit einem Konstruktor desselben Typs beginnen, sucht `matchCon` eine Liste aller Konstruktoren dieses Typs zusammen mit ihrer Arität. Die Konstruktorlisten werden im Zustand mitgeführt und können daraus ausgelesen werden. Die Funktion `matchClause` wird für jeden Konstruktor `c` einmal aufgerufen, wobei die Liste der übergebenen Gleichungen nur Gleichungen enthält, die mit `c` beginnen. Dadurch werden Gleichungen mit gleichem Konstruktor zusammen gruppiert und die ursprüngliche Reihenfolge der Gleichungen verändert. Dies ist aber – wie im Abschnitt 3.1.2 erwähnt – unproblematisch. Der Rückgabewert von `matchClause` ist dann ein Case-Ausdruck, bei dem die erste freie Variable aus der Variablenliste das Argument ist und die durch `matchClause` überarbeiteten Klauseln die Alternativen sind.

In `matchClause` selbst wird die Arität des Konstruktors `c`, mit dem `matchClause` aufgerufen wurde, benötigt. Diese kann durch Suchen in der mitgeführten Liste von Konstruktoren gefunden werden. Es müssen hier für jede Stelle des Konstruktors neue freie Variablen generiert werden, die dann vorne in die Liste der alten freien Variablen geschrieben werden bei dem match-Aufruf in `matchClause`. Gleichzeitig werden vorne in die Liste der Patterns jeder Gleichung die Patterns der dazugehörigen Kontruktoren geschrieben. Zurückgegeben wird eine Klausel, wobei das `CPattern` der Konstruktor mit den neuen freien Variablen als Parameter ist. Die `CRhs` ist eine `CSimpleRhs` mit dem durch den neuen match-Aufruf entstandenen Ausdruck und keinen lokalen Deklarationen.

In der Funktion

```
matchLit :: [CVarIName] -> [([CPattern], CExpr)]
         -> CExpr -> PMM CExpr
```

ist die Literal-Regel implementiert und es wird hier ein Case-Ausdruck in eine verschachtelte `if-then-else` Abfrage umgewandelt. Dies ist nötig, da Literale nicht wie ein Konstruktor behandelt werden können. Der Grund hierfür ist, dass beim Konstruktor alle Alternativen Konstruktoren, also die desselben Typs, ihre entsprechenden

Gleichungen zugeteilt bekommen. Das ist auch der Fall für Konstruktoren, die keine Gleichungen haben, da diese match-Aufrufe dann zum Default-Ausdruck ausgewertet werden. Weil Literale aber unendlich sind, können nicht für unendlich viele Alternativen match-Aufrufe getätigt werden.

Da der Algorithmus später nur auf Case-Ausdrücke angewandt werden soll, bezieht sich auch die Umsetzung der Literal-Regel nur auf die Behandlung von Case-Ausdrücken. Es wird damit dann ein Ausdruck der Form:

```
case e of
  l1 -> e1
  ⋮
  ln -> en
  x  -> e'
```

mit $l_1, \ldots, l_n$ Literale, `x` Variable und `e`, $e_1, \ldots,$, e_n und `e'` Ausdrücken zu einem Ausdruck der Form:

```
let x = e in
if x == l1
  then e1
  else if x == ...
  else if x == ln
         then en
         else e'
```

umgewandelt. Dies hat dieselbe Funktionaliät wie der dazugehörige Case-Ausdruck, da die Bedingungen für `e` in derselben Reihenfolge überprüft werden. Der `let` Ausdruck dient dazu, dass ein potenziell größerer Ausdruck `e` nicht für jede `if-then-else`-Abfrage vervielfältigt wird. Sollte die letzte Alternative mit `x -> e'` fehlen, so wird `e'` in dem umgeformten Ausdruck durch den Default-Ausdruck ersetzt. Falls nach der Variablen `x` im Case-Ausdruck noch weitere Literal-Pattern folgen, so werden diese nach der Umformung vergessen, sie sind also nicht mehr vorhanden. Da der Case-Ausdruck der Reihenfolge nach ausgewertet wird, sind weitere Literal-Pattern nach einer Variablen nicht erreichbar. Denn durch das Pattern-Matching wird `e` zu der ersten freien Variablen ausgewertet, womit dieser Fall gewählt wird statt eines späteren Falls, der genau zu `e` passt.

Die Implementierung der in diesem Kapitel beschriebenen match-Funktionen sind in Anhang A aufgeführt.

3.2.2. Anwendung auf ein Curry-Programm

Nachdem der Algorithmus implementiert ist, soll dieser auf Case-Ausdrücke in einem Curry-Programm eingesetzt werden. Dazu müssen alle Case-Ausdrücke im Code gefunden und die Funktion match auf diese angewandt werden.

Zuvor muss aber noch die im vorherigen Abschnitt angesprochene `TypeList` mit den Konstruktoren erstellt werden. Die Datentypen mit ihren Konstruktoren befinden sich alle in der Liste der `CTypeDecl` eines `CurryProg`. Für das zu überarbeitende Modul und alle importierten Module muss also diese Liste durchgegangen werden und für jeden Konstruktor die `TypeList` erstellt werden. Da die Konstruktoren bereits nach Typen in den `CTypeDecl` gruppiert sind, müssen also nur die `QName` jedes Konstruktors zu allen `QName` der Konstruktoren desselben Typs und ihrer Arität gruppiert werden. Damit sehen z.B. die Paare für die Listen-Konstruktoren `[]` und `:` wie folgt aus:

```
(("Prelude", "[]"), [(("Prelude", "[]"), 0), (("Prelude", ":"), 2)])
(("Prelude", ":"), [(("Prelude", "[]"), 0), (("Prelude", ":"), 2)])
```

Durch diese Struktur finden sich somit zu jedem Konstruktor alle anderen Konstruktoren des Typs. Da die Konstruktoren ihre Arität nicht direkt als Zahl mitführen, muss diese durch Zählen der `CTypeExpr` jedes Konstruktors erstellt werden. Sobald dieser Schritt vollzogen ist, kann match mit dieser `TypeList` im Zustand aufgerufen werden und das Programm nach Case-Ausdrücken durchsucht werden.

Die hier relevanten Case-Ausdrücke können alle in den Funktionsdeklarationen des AbstractCurry-Programms gefunden werden. Die Funktionsdeklarationen werden nach `CExpr` durchsucht, die `CCase` vom Typ `CRigid` sind. Nur auf diese Ausdrücke wird match angewandt, alle anderen Funktionen oder Funktionsteile, die keine Case-Ausdrücke enthalten, bleiben unverändert.

Die Funktion match, wie sie hier implementiert ist, arbeitet allerdings nur mit Paaren der Form `([CPattern], CExpr)` und die rechten Seiten eines Case-Ausdrucks sind vom Typ `(CPattern, CRhs)`. Diese rechten Seiten müssen damit so umgeschrieben werden, dass sie das richtige Format erhalten und keine Informationen wie lokale Deklarationen oder Guards verloren gehen. Die `CPattern` aus den rechten Seiten werden dafür einfach in einelementige Listen geschrieben. Um an den entsprechenden Ausdruck der rechten Seite bei einem `CSimpleRhs` zu kommen, reicht es bei fehlenden lokalen Deklarationen aus, einfach den Ausdruck der `CSimpleRhs` zu übernehmen. Lokale Deklarationen der `CRhs` sind immer im `where` Teil der Funktion. Damit nach der Transformation die Deklarationen erhalten bleiben, muss eine rechte Seite `e` der Form

```
e = r where decls
```

zu einem Ausdruck `e'` der Form

```
e' = let decls in r
```

umgeformt werden. In AbstractCurry ist ein Let-Ausdruck durch den Konstruktor `CLetDecl :: [CLocalDecl] -> CExpr -> CExpr` dargestellt, womit also durch Verschieben der `CLocalDecl` der für `match` benötigte Ausdruck erstellt ist.

Bei einem `CGuardedRhs` erfolgt der Erhalt der lokalen Deklarationen auf die gleiche Weise, allerdings müssen zunächst die Guards mit entsprechenden rechten Seiten von der Form `[(CExpr, CExpr)]` zu einem äquivalenten `CExpr` umgewandelt werden. Ein Ausdruck der Form

```
case e of
  p1 | c1 -> e1
     | c2 -> e2
  p2      -> e3
```

hat die Guards c_1 und c_2 mit den dazugehörigen Ausdrücken e_1 und e_2. Um dies in einen einzigen Ausdruck umzuwandeln, soll ein solcher Case-Ausdruck die Form

```
let x = e
in case x of
     p1 -> if c1 then e1
                 else if c2 then e2
                             else case x of
                                    p2 -> e3
     p2 -> e3
```

erhalten, womit alle Guards in einem `if`-Ausdruck untergebracht werden. Dieser neue Ausdruck ist äquivalent zu dem Alten, da die verschachtelten `if`-Abfragen die gleiche Reihenfolge haben, wie die Guards vorher. Sollte keine der `if`-Abfragen erfüllt werden, ist der Alternativfall so wie bei einem Case-Ausdruck mit Guards der Rest des alten Case-Ausdrucks.

Sobald ein Case-Ausdruck im Code gefunden wird und passend umgeformt ist, wird die Funktion `match` mit nur einer freien Variablen `pm_u1` aufgerufen, weil ein Case-Ausdruck nur einen Ausdruck hat, den er überprüft. Da der Case-Ausdruck aber normalisiert werden soll, muss eine Variable überprüft werden. Dazu wird vor dem Case-Ausdruck aus dem Quellprogramm `pm_u1` an den überprüften Ausdruck mittels eines `let`-Ausdrucks gebunden. Sollte es sich bei dem überprüften Ausdruck schon um

eine Variable handeln, wird nach der Normalisierung diese `let` Deklaration entfernt und jedes Vorkommen von `pm_u1` durch die alte Variable substituiert.

Wird die Funktion `match` nun auf eine Funktion mit Case-Ausdruck angewandt, so wird diese normalisiert. Betrachten wir als Beispiel die Funktion `det` aus Abschnitt 2.1.5. Diese wird wie folgt umgewandelt:

```
det x =
  case x of
    [] -> 0
    pm_u2 : pm_u3 -> 42
```

Wie schon bei der Einführung von `det` erwähnt, muss zur Normalisierung nur der Unterstrich durch den Listenkonstruktor mit zwei Variablen ersetzt werden. Die Variablen `pm_u2` und `pm_u3` sind durch die Variablen-Regel erstellt worden. Sie werden nicht weiter benötigt, womit sie auch selbst Unterstriche sein können, aber das ist bei der Generierung der Variablen nicht unbedingt klar. Zu sehen ist hier auch, dass die Variable `pm_u1` fehlt. Da diese zunächst an den Ausdruck `x` gebunden wurde und dieser eine Variable ist, wurde der enstprechende `let` Ausdruck wie oben erklärt entfernt.

Die folgende Funktion `det2` überprüft ein Paar von Variablen:

```
det2 x y = case (x,y) of
             ([], []) -> 0
             _        -> 42
```

Hier steht der Unterstrich nicht nur für einen anderen Fall, sondern für die Fälle, dass `x` und `y` leer sind, dass nur `x` leer ist aber `y` nicht und dass `x` nicht leer und `y` beliebig ist. Durch die Funktion `match` wird dies wie folgt normalisiert:

```
det2 x y =
  let pm_u1 = (x,y)
  in case pm_u1 of
       (pm_u2,pm_u3) ->
         case pm_u2 of
           [] ->
             case pm_u3 of
               [] -> 0
               pm_u4 : pm_u5 -> 42
           pm_u6 : pm_u7 -> 42
```

Alle Case-Ausdrücke hier sind normalisiert und auch die Variable `pm_u1` ist erhalten geblieben. Funktional macht die Funktion auch noch das Gleiche, da sie nur bei zwei leeren Liste `0` zurückgibt und sonst `42`.

4. Optimierungen

Die von `match` erstellten Ausdrücke können zu unoptimiertem Code führen. In dieser Arbeit werden zwei mögliche Optimierungen betrachtet, die zu einer Verringerung der Codegröße führen können. Zum einen kann es zu doppelten Überprüfungen eines Ausdrucks innerhalb eines Case-Ausdrucks kommen. Ein Beispiel für eine Funktion, bei der dies stattfindet, ist in Listing 4.1 zu finden. Dabei sind `a`, `b` und `c` hier nicht relevante

```
demo :: [a] -> [a] -> [a]
demo xs ys = case (xs, ys) of
  ([], ys')       -> a ys'
  (xs', [])       -> b xs'
  (x:xs', y:ys') -> c x xs' y ys'
```

Listing 4.1.: Beispielfunktion `demo`

Funktionen. Bei dieser Funktion muss die Mischungs-Regel angewandt werden, da das erste Pattern mit einem Konstruktor `[]` beginnt, das zweite Pattern mit einer Variablen `xs'` und das dritte wieder mit einem Konstruktor `(:)`. Das heißt, der Ausdruck wird in drei Gruppen eingeteilt. Beim ersten wird durch die Konstruktor-Regel überprüft, ob es sich um eine leere Liste oder nicht handelt. Sollte es sich nicht um eine handeln, so wird die zweite Liste überprüft. Hier werden wegen der Mischungs-Regel zwei Gruppen gemacht, wo geschaut wird ob ys die leere Liste `[]` oder eine nicht-leere Liste mit `(:)` ist. Durch diese Verschachtelung entstehen doppelte Überprüfungen von `xs` und `ys`. Der Code in Listing 4.2 ist durch den Aufruf von `match` aus der Funktion `demo` entstanden. Die Variablen `xs` und `ys` sind durch `pm_u2` und `pm_u3` substituiert worden. Es ist zu erkennen, dass zunächst `pm_u2` überprüft wird und dann `pm_u3`. Durch diese beiden Überpüfungen sind die ersten beiden Fälle des Case-Ausdrucks abgedeckt. Wegen der Gruppierung bei den ersten Aufrufen werden nun aber für den letzten Fall, bei dem keine der Listen leer sein dürfen, `pm_u2` und `pm_u3` erneut überprüft. Sollte es sich hier doch um leere Listen handeln, so wird der Default-Ausdruck, hier also `failed`, zurückgegeben, ansonsten der gewünschte Aufruf. Der Fall der leeren Liste kann allerdings nie auftreten, da diese Stelle im Code nur erreicht werden kann,

```
demo xs ys =
  let pm_u1 = (xs,ys)
  in case pm_u1 of
       (pm_u2,pm_u3) ->
         case pm_u2 of
           [] -> a pm_u3
           pm_u10 : pm_u11 ->
             case pm_u3 of
               [] -> b pm_u2
               pm_u8 : pm_u9 ->
                 case pm_u2 of
                   [] -> failed
                   pm_u4 : pm_u5 ->
                     case pm_u3 of
                       [] -> failed
                       pm_u6 : pm_u7 -> c pm_u4 pm_u5 pm_u6 pm_u7
```

Listing 4.2.: Normalisierte Funktion `demo`

wenn `pm_u2` und `pm_u3` nicht-leer sind. Das heißt man kann diese Überprüfungen entfernen und direkt statt des zweiten `case pm_u2 of ...` die rechte Seite des letzten Falles zurück geben. Damit würde sich diese Funktion auf den Code in Listing 4.3 reduzieren lassen. Erreicht wird diese Verbesserung, nachdem der Case-Ausdruck mit

```
demo xs ys =
  let pm_u1 = (xs,ys)
  in case pm_u1 of
       (pm_u2,pm_u3) ->
         case pm_u2 of
           [] -> a pm_u3
           pm_u10 : pm_u11 ->
             case pm_u3 of
               [] -> b pm_u2
               pm_u8 : pm_u9 -> c pm_u10 pm_u11 pm_u8 pm_u9
```

Listing 4.3.: Normalisierte und optimierte Funktion `demo`

doppelten Überprüfungen von `match` erstellt ist. Der Ausdruck wird dann überprüft, wobei für jedes erste Vorkommen von Variablen in den Pattern protokolliert wird, zu welchem Konstruktor diese Variable ausgewertet wird. Sollte eine Variable ein zweites Mal in diesem Case-Ausdruck durch einen verschachtelten Case-Ausdruck wie

im oberen Beispiel überprüft werden, so wird die Überprüfung durch die rechte Seite des dazugehörigen Patterns ersetzt. Die entsprechenden Variablen des ersetzten Patterns müssen dann durch die Variablen des protokollierten Patterns substituiert werden.

Bei dem oberen Beispiel wird also zunächst das zweite Vorkommen von `case pm_u2 of` durch

```
case pm_u3 of
  [] -> failed
  pm_u6 : pm_u7 -> c f pm_u10 pm_u11 pm_u6 pm_u7
```

ersetzt und da dies auch das zweite Vorkommen von `case pm_u3 of` ist, wird dieser Ausdruck durch

```
c f pm_u10 pm_u11 pm_u8 pm_u9
```

ersetzt. Hierbei wurden zunächst `pm_u4` und `pm_u5` durch `pm_u10` und `pm_u11` substituiert und anschließend auch noch `pm_u6` und `pm_u7` durch `pm_u8` und `pm_u9`. Mit dem Protokollieren der bereits sicher ausgewerteten Variablen kann der Code in solchen Fällen also um einiges verringert werden, besonders dann, wenn es mehr Konstruktoren als nur zwei wie in diesem Beispiel gibt.

Das zweite Problem, das bei diesem Algorithmus auftreten kann, ist die Verdoppelung von großen Codeblöcken. Dies kann bei Anwendung der Mischungs-Regel passieren, genauer bei der danach verwendeten Konstruktor-Regel, da hier `match` für alle Konstruktoren eines Typs aufgerufen wird, selbst wenn es keine Gleichungen für manche Konstruktoren gibt. Ein einfaches Beispiel ist hier Folgendes:

```
data Numbers = One Int | Two Int Int | Three Int Int Int

f :: Numbers -> Int
f x = case x of
        One y -> y
        _     -> e
```

Dabei bezeichnet `e` einen beliebigen Ausdruck, der einen `Int` zurückgibt. Sollte `match` auf diese Funktion angewandt werden, so ergibt sich der nachstehende Aufruf:

```
match [x]
      [([One y], y),
       ([_],    e)]
      def
```

Da das erste Pattern mit einem Konstruktor und das zweite Pattern mit einer Variablen beginnt, muss die Mischungs-Regel angewandt werden, womit verschachtelte match-Aufrufe enstehen. Der obere Aufruf wird zu dem Aufruf

```
match [x]
      [([One y], y)]
      (match [x]
             [([_], e)]
             def)
```

umgewandelt, die Gleichungen sind somit aufgeteilt worden. Der innere match-Aufruf wird durch die Variablen-Regel zu `e` ausgewertet, womit der äußere als Default-Ausdruck `e` hat:

```
match [x]
      [([One y], y)]
      e
```

Bei diesem match-Aufruf wird die Konstruktor-Regel angewandt, womit der folgende Case-Ausdruck entsteht:

```
case x of
  One pm_u2 -> match [pm_u2] [([y], y)] e
  Two pm_u3 pm_u4 -> match [pm_u3, pm_u4] [] e
  Three pm_u5 pm_u6 pm_u7 -> match [pm_u5, pm_u6, pm_u7] [] e
```

Im ersten match-Aufruf wird die Variable Rule angewandt, `y` durch `pm_u2` substituiert und der gesamte Ausdruck schließlich zu `pm_u2` ausgewertet. Da bei den anderen beiden match-Aufrufen keine Gleichungen vorhanden sind, wird so lange die Variablen-Regel angewandt, bis alle Variablen aus der Variablen-Liste entfernt wurden und somit keine Variablen und Gleichungen noch vorhanden sind. Dadurch werden diese beiden Aufrufe zum Default-Ausdruck ausgewertet, in diesem Fall also `e`. Nach Anwendung des Algorithmus sieht die Funktion `f` folgendermaßen aus:

```
f x =
  case x of
    One pm_u2 -> pm_u2
    Two pm_u3 pm_u4 -> e
    Three pm_u5 pm_u6 pm_u7 -> e
```

Wegen der drei Konstruktoren von `Numbers`, von denen nur einer mit einer Gleichung im Case-Ausdruck abgedeckt ist, muss der alternative Fall mit `e` also für `Two` und `Three` beim Matching geschrieben werden. Dadurch ist der Ausdruck `e` verdoppelt worden. Sollte `e` ein großer Ausdruck – wie zum Beispiel ein verschachtelter Case-Ausdruck – sein, so wird der gesamte Code von `f` sehr viel größer. Durch das Normalisieren in den Case-Ausdrücken lässt sich die Verdoppelung dieser rechten Seiten auch nicht verhindern, da jeder Fall abgedeckt sein muss. Das bedeutet es muss verändert werden, was genau verdoppelt wird. Es können die Ausdrücke verdoppelt werden, die durch das Aufteilen der Gleichungen als neuer Default-Ausdruck an einen verschachtelten match-Aufruf übergeben werden. Dies tritt also bei Benutzung der Mischungs-Regel auf, wie in dem Beispiel oben.

Um nun zu verhindern, dass der möglicherweise große Ausdruck `e` dupliziert wird, wird er als lokale Deklaration in einem `let`-Ausdruck an eine neue frische Variable gebunden. Sofern die rechte Seite groß genug ist – abhängig von der an den Zustand übergebenen Größe – wird der `let`-Ausdruck mit der gerade generierten lokalen Deklaration und dem neuen match-Aufruf erstellt. Der Default-Ausdruck dieses match ist dann die neu erstellte Variable. Für die obere Funktion `f` sieht die Optimierung dann also wie folgt aus:

```
f x =
  let pm_u2 = e
  in case x of
       One pm_u3 -> pm_u3
       Two pm_u4 pm_u5 -> pm_u2
       Three pm_u6 pm_u7 pm_u8 -> pm_u2
```

Hier wird nur noch die neue Variable `pm_u2` verdoppelt, wohingegen der Ausdruck `e` nur noch einmal vorkommt.

Es wird aber nicht zwangsläufig jeder neue Default-Ausdruck verdoppelt. Wenn zum Beispiel der Datentyp `Numbers` nur die ersten beiden Konstruktoren hätte, würde der Code hier dann sogar etwas größer als ohne die Optimierung, da ein neuer `let`-Ausdruck mit einer Variablen hinzukommt, ohne dass diese Variable benötigt wird. Um dies zu verhindern, muss getestet werden, ob die neue lokale Variable überhaupt mehrfach auf der rechten Seite vorkommt. Wenn das wie oben der Fall ist, wird der `let`-Ausdruck erstellt. Wenn nicht, dann wird das Vorkommen der lokalen Variablen durch den alten Default-Ausdruck substituiert und das `let` gar nicht erst erstellt.

In der Implementierung kann mit dem zweiten Flag im Zustand angegeben werden, ob die Optimierung verwendet werden soll und ab welcher Größe von rechten Seiten. Falls der alte Default-Ausdruck ebenfalls nur eine Variable oder ähnliche kleiner Ausdruck ist, so ist zum Beispiel keine neue lokale Variable nötig.

5. Auswertung des Algorithmus

Dieses Kapitel behandelt die Auswertung der Umsetzung des Algorithmus. Dabei wird der Algorithmus mit und ohne Optimierungen auf verschiedene Testprogramme angewandt. Anschließend wird von dem überarbeiteten AbstractCurry Programm die Größe aller Ausdrücke berechnet, indem für jedes `CExpr`, `CPattern`, etc. die Anzahl der Ausdrücke gezählt wird. Jeder Ausdruck hat dabei an sich die Größe eins und darauf addiert werden die Werte der Unterausdrücke.

Zum Beispiel hat ein `CVar :: CVarIName -> CExpr` also die Größe eins, da eine Variable keine Unterausdrücke enthält. Bei einem Ausdruck mit möglichen Unterausdrücken wie ein `let`-Ausdruck mit `CLetDecl :: [CLocalDecl] -> CExpr -> CExpr` ist die Größe eins, plus die Größe der Summe seiner lokalen Deklarationen und die Größe seines `in`-Ausdrucks. Es werden also alle Ausdrücke erfasst und gezählt und die Gesamtsumme gibt somit einen groben Überblick über die Größe des neuen Programms. Hiermit kann gemessen werden, wie viel die einzelnen Optimierungen alleine oder auch zusammen die Codegröße verringern oder ob sie vielleicht sogar zu mehr Code führen können.

In der folgenden Tabelle sind einige Programmgrößen nach Anwendung des Algorithmus zu finden. Die Testprogramme sind in Curry an sich nicht sehr groß, allerdings ist die Größe des Programms vor Anwendung und nach Anwendung nicht gut zu vergleichen, da normalisierte Case-Ausdrücke häufig größer sind:

Als Mindestgröße für rechte Seiten, die bei Code-Duplizierung durch eine lokale Variable ersetzt werden sollen, wurde willkürlich fünf gewählt. Es ist bei p_1 und p_2 zu sehen, dass die Optimierungen keinerlei Auswirkung auf die Größe des Programms haben. Hier werden also schon vor Optimierung keine Variablen doppelt überprüft oder rechte Seiten größer als fünf verdoppelt. Bei p_3 und p_4 wirkt sich nur das Wegoptimieren der doppelten Variablenüberprüfung auf die Programmgröße aus. Diese bewirkt etwas bei den beiden Programmen, da in p_3 die `demo` Funktion aus dem vorherigen Abschnitt implementiert und in p_4 ein Case-Ausdruck mit Guards vorhanden ist. Damit entstehen bei beiden verschachtelte Case-Ausdrücke, die dieselbe Variable als Parameter haben. Da der innere Case-Ausdruck, der beim Wegtransformieren des Guards entsteht, auch

	Codegröße nach Optimierung			
Programm	Keine	Doppeltes Überprüfen	Code-Duplizierung	Beide
p_1	123	123	123	123
p_2	118	118	118	118
p_3	131	117	131	117
p_4	54	37	54	37
p_5	79	79	45	45
p_6	1103	1103	93	93
p_7	263	207	161	154
p_8	3943	3938	3087	3082

Tabelle 5.1.: Anzahl der Ausdrücke in den Programmen p_1 bis p_8 nach Anwendung des Algorithmus mit verschiedenen Optimierungen

normalisiert sein muss, kann hier je nach Wahl der restlichen Patterns durch die Optimierung der letzte `else` Fall kein Case-Ausdruck mehr sein. Sollten die Patterns sich bei dem Case-Ausdruck mit Guards nicht überlappen, wird dieser Fall wegoptimiert.

In p_1 bis p_4 hat es keine Code-Duplizierung gegeben, da die Funktionen in diesen Programmen alle nur mit Listen oder Literalen als Datentypen gearbeitet haben. Damit wurde also jede rechte Seite der Gleichungen, die nur eine Variable als Pattern haben, entweder durch den fehlenden Konstruktor ersetzt oder durch einen `if-then-else` Ausdruck. In p_5 und p_6 wurde daher ein eigener Datentyp mit mehr als zwei Konstruktoren verwendet. Beide Programme enthalten lediglich dieselbe folgende Funktion:

```
unwieldy :: TestData -> TestData -> Int
unwieldy x y = case x of
                 One _ -> 9999999
                 _     -> case y of
                            Two _ _ -> 3
                            _       -> let x = 1
                                           y = 6
                                       in x + y
```

In p_5 hat der Datentyp `TestData` allerdings lediglich die beiden Konstruktoren `One`, `Two` und noch einen Zusätzlichen, wohingegen er in p_6 die beiden ersten Konstruktoren und noch acht weitere Konstruktoren enthält. Die Funktion ist extra so konstruiert, dass möglichst viel Duplikation auftritt. So muss der innere Case-Ausdruck für jeden Konstruktor außer `One` dupliziert werden und der `let`-Ausdruck im Inneren für alle außer `Two`. Wie in der Tabelle zu erkennen, gibt es auch einen sehr großen Unterschied bei

der Anwendung des Algorithmus. Ohne die Optimierung ist p_6 mit sieben zusätzlichen Konstruktoren ungefähr vierzehn mal so groß wie p_5. Mit Optimierung ist es nur noch knapp doppelt so groß. Vergleicht man zudem die Größenunterschiede zwischen ohne Optimierung und mit, so ist deutlich zu erkennen, dass die Optimierung sich sehr viel stärker auswirkt, je mehr Konstruktoren der in den Case-Ausdrücken überprüfte Datentyp hat.

Das Programm p_7 enthält eine leicht modifizierte Version von `demo`, die nun mit Listen von einem Datentypen mit vier Konstruktoren arbeitet. Dadurch können beide Optimierungen angewandt werden, was sich auch in den Zahlen in Tabelle 5.1 wiederspiegelt. Allerdings ist hier auch zu erkennen, dass sich die beiden Optimierungen auf Grund der Art ihrer Implementierung gegenseitig etwas behindern können. So bringt die Entfernung von doppelter Code-Überprüfung alleine eine Größenverringerung von ungefähr sechzig, wird aber zusätzlich noch die Code-Duplizierung entfernt, so ist der Größenunterschied nur sieben. Das liegt daran, dass die Vermeidung der Code Duplizierung zuerst stattfindet und somit die Case-Ausdrücke, die doppelt überprüft werden, in eine lokale Variable geschrieben werden. Sobald am Ende des Algorithmus also diese Fälle entfernt werden, sind zwei verschiedene Case-Ausdrücke vorhanden, da einer in einer Variablen endet. Es werden also zwei Case-Ausdrücke separat voneinander überprüft. Dabei kann die Optimierung nicht erkennen, dass durch die lokale Variable verschachtelte Case-Ausdrücke entstehen.

Da die Programme p_1 bis p_7 alles konstruierte Beispiele sind, um bestimmte Funktionen des Algorithmus zu testen, wurde der Algorithmus zusätzlich noch auf ein weiteres Programm p_8 angewandt. Dieses Programm ist die Implementierung des Algorithmus selbst. Hierbei hat die Entfernung doppelter Überprüfung kaum eine Auswirkung auf die Gesamtgröße, da in beiden Fällen nur fünf Ausdrücke eingespart werden. Die Entfernung von Code-Duplizierung hingegen hat sehr große Auswirkung mit einer Verringerung um beinahe 900 Ausdrücke. Der normalisierte Programmcode des Algorithmus erzeugt bei Anwendung auf die Programme der Tabelle bei gleichen Parametern den gleichen Programmcode. Hiermit ist gezeigt, dass hier die Funktionaliät durch den Algorithmus nicht geändert wurde.

Weiterhin hat auch das doppelte Anwenden des Algorithmus auf ein Programm keine Auswirkung auf die Größe, da die Normalisierung bereits stattgefunden hat. Wird der Algorithmus zum Beispiel auf die überarbeitete unoptimierte Variante von p_8 mit einer Größe von 3943 angewandt, so wird das gleiche Programm mit derselben Größe erstellt. Es werden also keine weiteren Änderungen vorgenommen.

Bei den Tests ist aufgefallen, dass die Entfernung von doppelter Überprüfung nur zu einem kleineren oder gleich großem Programm führen kann. Das liegt in der Art der Implementierung, da bei dieser Optimierung niemals Code hinzugefügt sondern nur entfernt wird. Folglich kann die Optimierung sich also niemals negativ auf die Codegröße auswirken und sollte somit immer mit benutzt werden. Die Optimierung kann sich einzig auf die Rechenzeit des Programms auswirken, da sie bei nicht vorhandener doppelter Überprüfung bei jeder Anwendung von `match` aufgerufen wird, ohne Auswirkungen zu haben.

In Tabelle 5.1 gibt es für die Entfernung von Code-Duplizierung nur Beispiele, die wie bei der anderen Optimierung zu einer höchstens positiven Änderung der Codegröße führen. Dies liegt an der Wahl der Größe fünf für rechte Seiten. Durch die Benutzung von anderen Größen kann die Optimierung auch zu größerem Code führen. Bei p_2 beispielsweise führt eine Größe von eins zu einer gesamten Codegröße von 121 statt 118. Hier werden rechte Seiten der Größe eins durch eine Variable der Größe eins ersetzt und zusätzlich noch der `let`-Ausdruck hinzugefügt, wodurch im Endeffekt mehr Code ensteht. Bei p_1 führen aber auch die Größe zwei und drei zu mehr Code, allerdings nur zu einer Gesamtgröße von 124 statt 123. Der Grund hierfür ist wieder, dass durch den `let`-Ausdruck etwas größerer Code hinzugefügt wird, als durch Einsetzen von Variablen auf der rechten Seite eingespart wird.

Das bedeutet, dass das bei einer zu klein gewählten Größe für die Optimierung leicht größerer Code enstehen kann. Wählt man die Größe aber zu groß, so kann es ebenfalls wieder zu mehr Code führen, wie bei p_5 und p_6. Bis zur Größe zehn bleibt die gesamte Codegröße – wie in bei Größe fünf – bei 45 beziehungsweise 93 Ausdrücken. Sobald aber die Größe elf gewählt wird, steigt sie auf je 51 und 162 an. Es gibt in diesen Programmen also rechte Seiten der Größe zehn, die mehrfach dupliziert werden. Sobald diese nicht mehr durch eine Variable ersetzt werden, wird natürlich auch der Code größer. Die Auswirkung ist bei p_6 größer als bei p_5, da hier durch mehr Konstruktoren im Datentyp auch mehr Duplizierung auftritt.

Die Größe darf für eine möglichst geringe Codegröße nicht zu groß und nicht zu klein gewählt werden. Da es aber schwer ist die optimale Größe vorausszusehen und diese sich auch von Programm zu Programm unterscheiden kann, ist es demnach von Vorteil eine möglichst kleine Größe zu wählen, die allerdings nicht eins ist. Wie eben erwähnt kann es hierbei auch zu mehr Code führen, aber die Auswirkung auf die Gesamtgröße ist sehr gering. Wird die Größe jedoch zu groß gewählt, kann sich die Möglichkeit ergeben, dass bereits mehrfach duplizierte Codeblöcke weiterhin dupliziert werden und keine Optimierung stattfindet.

6. Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurde ein Algorithmus implementiert, der Case-Ausdrücke in Curry-Programmen normalisieren kann und zusätzlich noch einige Optimierungen beinhaltet, die zu einer geringeren Codegröße führen können. Dafür wurde ein bereits vorhandener Algorithmus [5] in Curry implementiert, so dass er mit Curry-Programmen auf der Ebene von AbstractCurry arbeiten kann. Dieser Algorithmus benutzt eine Funktion match, die Case-Ausdrücke durch Verwendung verschiedener Regeln normalisiert.

Die manchmal auftretenden Probleme der doppelten Überprüfung von Variablen und der Code-Duplizierung wurden anschließend durch Implementierung von Optimierungen für den Algorithmus behoben. Dabei findet die Optimierung der doppelten Überprüfung nach Anwendung von match und die Verhinderung der Code-Duplizierung währenddessen statt.

Durch das Zählen von Ausdrücken in Curry-Programmen mit normalisierten Case-Ausdrücken wurde gezeigt, dass die Optimierungen unterschiedlich starke Auswirkung auf die Codegröße haben. Die Entfernung von Code-Duplizierung hat sich hier sehr viel mehr auf die Codegröße ausgewirkt, als die Entfernung der doppelten Überprüfung. Dabei wurde aber auch festgestellt, dass die Entfernung von Code-Duplizierung die Anwendung der anderen Optimierung durch Erstellung lokaler Deklarationen behindern kann.

Um zukünftig normalisierte Case-Ausdrücke mit möglichst geringer Codegröße zu erhalten, kann überlegt werden, die Implementierung zu überarbeiten. Dabei kann man versuchen, die Entfernung der doppelten Überprüfung von Variablen so zu überarbeiten, dass sie verschachtelte Case-Ausdrücke auch bei Benutzung von lokalen Variablen noch optimieren kann.

Alternativ kann auch ein anderer Pattern-Matching-Algorithmus benutzt werden, um Case-Ausdrücke zu normalisieren, bei dem die doppelte Überprüfung gar nicht erst entstehen kann. Ein solcher Algorithmus lässt sich z.B. in [4] finden. Dieser Algorithmus hat jedoch noch das Problem der Code-Duplizierung, so dass der hier erarbeitete Optimierungsansatz dort implementiert werden müsste.

A. Auszüge der Implementierung

In diesem Kapitel sind Auszüge der in dieser Arbeit erstellten Implementierung enthalten. Diese Auszüge beinhalten die match-Funktionen und die meisten von ihnen benutzten Funktionen, sowie die Optimierungen. Die Entfernung der doppelten Überprüfung ist dabei in der Funktion `optimize` implementiert und die Entfernung von Code-Duplizierung in der Funktion `match` selbst. Der in dieser Arbeit beschriebene Stand der Implementierung entspricht der Version im verwendeten Arbeitsrepository mit dem Tag `abgabe`.

```
--- Determines if an equation begins with a variable
isVar :: ([CPattern], CExpr) -> Bool
isVar (ps, _) = case ps of
  CPVar     _ : _ -> True
  CPComb _ _ : _ -> False
  CPLit     _ : _ -> False
  _              -> error $ "isVar: " ++ (show ps)

--- Determines if an equation begins with a constructor
isCon :: ([CPattern], CExpr) -> Bool
isCon (ps, _) = case ps of
  CPVar     _ : _ -> False
  CPComb _ _ : _ -> True
  CPLit     _ : _ -> False
  _              -> error $ "isCon: " ++ (show ps)

--- Determines if an equation begins with a literal
isLit :: ([CPattern], CExpr) -> Bool
isLit (ps, _) = case ps of
  CPVar     _ : _ -> False
  CPComb _ _ : _ -> False
  CPLit     _ : _ -> True
  _              -> error $ "isLit: " ++ (show ps)

--- Determines if an equation begins with an as-pattern
isAs :: ([CPattern], CExpr) -> Bool
isAs (ps, _) = case ps of
```

```
  CPAs _ _ : _ -> True
  _            -> False

--- Returns a constructor an equation begin with
getCon :: ([CPattern], CExpr) -> QName
getCon ps = case ps of
  (((CPComb c ps') :_), _) -> c
  _                        -> error "getCon"

--- Create a number for a fresh variable
freshVar :: PMM Int
freshVar = getsS freshIdx >+= \v -> modifyS (\(SipeyState i c f1 f2) ->
                                    (SipeyState (i + 1) c f1 f2)) >+ returnS v

--- Return a new variable with a number created by freshVar
makeVar :: PMM CVarIName
makeVar =
  freshVar >+= \i ->
  returnS (1, varPrefix ++ show i)  -- index is 1 for now, maybe adjust later

--- Prefix for variable construction
varPrefix :: String
varPrefix = "pm_u"

--- This function implements the variable rule
matchVar :: [CVarIName] -> [([CPattern], CExpr)] -> CExpr -> PMM CExpr
matchVar []     _ _   = error "matchVar"
matchVar (u:us) qs def =
  match us [(ps, substSingle v (CVar u) e)
           | (((CPVar v):ps), e) <- qs] def

--- This function implements the constructor rule
matchCon :: [CVarIName] -> [([CPattern], CExpr)] -> CExpr -> PMM CExpr
matchCon []     _        _   = error "matchCon"
matchCon (_:_)  []       _   = error "matchCon"
matchCon (u:us) qs@(q:_) def =
    consOfType (getCon q) >+= \cs ->
    mapS mClause cs >+= \mc ->
    returnS (CCase CRigid (CVar u) mc)
  where
    mClause (c, a) = matchClause c a (u:us) (choose c qs) def

--- Creates new variables for the patterns of a given constructor. Calls match
--- with these new patterns and variables added afterwards.
```

```
matchClause :: QName -> Int -> [CVarIName] -> [([CPattern], CExpr)] -> CExpr -> PMM (CPattern, CRhs)
matchClause _ _ []     _  _   = error "matchClause"
matchClause c a (_:us) qs def =
  replicateS a makeVar >+= \us' ->
  match (us' ++ us) [(ps' ++ ps, e) | (CPComb _ ps' :ps, e) <- qs] def >+= \mtch ->
  returnS (cBranch (CPComb c (map CPVar us')) mtch)

--- Returns all equations that begin with constructor c
choose :: QName -> [([CPattern], CExpr)] -> [([CPattern], CExpr)]
choose c qs = [q | q <- qs, getCon q == c]

--- This function handles match on literals
matchLit :: [CVarIName] -> [([CPattern], CExpr)] -> CExpr -> PMM CExpr
matchLit []       _        _   = error "matchLit"
matchLit (_:_)    []       _   = error "matchLit"
matchLit us@(_:_) qs@(_:_) def = makeIfThenElse us qs def

--- Creates an if_then_else expression from input
makeIfThenElse :: [CVarIName] -> [([CPattern], CExpr)] -> CExpr -> PMM CExpr
makeIfThenElse []     _  _   = error "makeifThenElse: no variables"
makeIfThenElse (u:us) qs def = case qs of
  [((CPLit l : ps), e)] ->
    match us [(ps, e)] def >+= \mtch ->
    returnS (applyF ("Prelude","if_then_else")
                    [applyF ("Prelude","==") [CVar u, CLit l], mtch, def])
  (((CPLit l : ps), e):q:qss) ->
    match us [(ps, e)] def >+= \mtch ->
    makeIfThenElse (u:us) (q:qss) def >+= \newIf ->
    returnS (applyF ("Prelude","if_then_else")
                    [applyF ("Prelude","==") [CVar u, CLit l], mtch, newIf])
  _ -> error $ "makeIfThenElse: " ++ show qs

--- Implementation of match, to generate case expressions. This covers the
--- empty rule and mixture rule. This version makes the use of matchVarCon
--- obsolete. This function can also remove code duplication
match :: [CVarIName] -> [([CPattern], CExpr)] -> CExpr -> PMM CExpr
match []       []    def = returnS def
match []       (q:_) _   = returnS (snd q)
match us@(u:_) qs    def
  | any isAs  qs = match us (map (removeAs u) qs) def
  | all isVar qs = matchVar us qs def
  | all isCon qs = matchCon us qs def
  | all isLit qs = matchLit us qs def
  | otherwise    = let (qs1, qs2) = splitSame qs
```

```
                in  match us qs2 def >+= \def' ->
                    getsS optCodeDuplication >+= \f2 ->
                    if (sizeExpr def') >= f2 && not (f2 == 0)
                      then makeVar >+= \v ->
                           match us qs1 (CVar v) >+= \mtch ->
                           returnS $ removeSingleLet (letExpr [CLocalPat (CPVar v) (CSimpleRhs def'
                      else match us qs1 def'

--- Removes as-patterns from the beginning of equations to return an equivalent
--- expression
removeAs :: CVarIName -> ([CPattern], CExpr) -> ([CPattern], CExpr)
removeAs _ ([],     _) = error "removeAs no Pattern"
removeAs u ((p:ps), e) = case p of
  CPAs v p' -> ((p':ps), substSingle v (CVar u) e)
  _         -> ((p:ps), e)

--- Splits lists of pattern-expression-pairs in two lists, where elements in the
--- first list are of the same expression type (Var, Con, Lit) and the second
--- list contains the rest
splitSame :: [([CPattern], CExpr)] -> ([([CPattern], CExpr)], [([CPattern], CExpr)])
splitSame []     = error "splitSame empty"
splitSame (q:qs) | isVar q   = splitVar qs [q]
                 | isCon q   = splitCon qs [q]
                 | isLit q   = splitLit qs [q]
                 | otherwise = error "There should be no split here"
  where
    splitVar []         _  = error "splitVar empty"
    splitVar yss@(y:ys) xs | isVar y            = splitVar ys (xs ++ [y])
                           | isCon y || isLit y = (xs, yss)
                           | otherwise          = error "splitVar"
    splitCon []         _  = error "splitCon empty"
    splitCon yss@(y:ys) xs | isCon y            = splitCon ys (xs ++ [y])
                           | isVar y || isLit y = (xs, yss)
                           | otherwise          = error "splitCon"
    splitLit []         _  = error "splitLit empty"
    splitLit yss@(y:ys) xs | isLit y            = splitLit ys (xs ++ [y])
                           | isVar y || isCon y = (xs, yss)
                           | otherwise          = error "splitLit"

--- Optimizes an expression returned by match to not double check variables in
--- case expressions
optimize :: CExpr -> CExpr
optimize = optExp []
  where
```

```
optExp :: [(CVarIName, CPattern)] -> CExpr -> CExpr
optExp _ v@(CVar _) = v
optExp _ l@(CLit _) = l
optExp _ s@(CSymbol _) = s
optExp r (CApply e1 e2) = CApply (optExp r e1) (optExp r e2)
optExp r (CLambda ps e) = CLambda ps (optExp r e)
optExp r (CLetDecl ls e) = CLetDecl (map (optLoc r) ls) (optExp r e)
optExp r (CDoExpr ss) = CDoExpr (map (optStat r) ss)
optExp r (CListComp e ss) = CListComp (optExp r e) (map (optStat r) ss)
optExp r (CCase t e prhs) = case e of
  CVar i -> case lookup i r of
    Nothing -> CCase t e [(p, (optRhs (extend r i p) rhs)) | (p, rhs) <- prhs]
    Just p -> optExp r (select p prhs)
  _      -> CCase t e [(p, optRhs r rhs) | (p, rhs) <- prhs]
optExp r (CTyped e te) = CTyped (optExp r e) te
optExp r (CRecConstr n1 fs) = CRecConstr n1 [(n2, optExp r e) | (n2, e) <- fs]
optExp r (CRecUpdate e1 fs) = CRecUpdate (optExp r e1) [(n, optExp r e2) | (n, e2) <- fs]

optRhs r (CSimpleRhs e ls) = CSimpleRhs (optExp r e) (map (optLoc r) ls)
optRhs _ (CGuardedRhs _ _) = error "optimize GuardedRhs"

optLoc r (CLocalPat p rhs) = CLocalPat p (optRhs r rhs)
optLoc _ l@(CLocalFunc _) = l
optLoc _ l@(CLocalVars _) = l

optStat r (CSExpr e) = CSExpr (optExp r e)
optStat r (CSPat p e) = CSPat p (optExp r e)
optStat r (CSLet ls) = CSLet (map (optLoc r) ls)

-- Adds a variable with the pattern it's been evaluated to
extend r i p = case p of
  CPComb _ _ -> (i, p) : r
  _          -> r

-- Selects the expression a that matches to constructor p.
-- Only gets called, when a variable has been evaluated to p
select p []        = error $ "select: " ++ show p
select p bss@(_:_) = case (p, bss) of
  (CPComb n ps, ((CPComb n' ps', CSimpleRhs e _) : bs))
    | n == n'   -> subst (mkSubst [v' | CPVar v' <- ps']
                                  [CVar v | CPVar v <- ps]) e
    | otherwise -> select p bs
  (_,_) -> error "error select"
```

Literatur

[1] S. Antoy und M. Hanus. *Curry: A Tutorial Introduction*. Available at `http://www.curry-language.org`. 2014.

[2] G. Barrett und P. Wadler. „Derivation of a pattern-matching compiler“. In: *Manuscript, Programming Research Group, Oxford* (1986).

[3] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-language.org`. 2016.

[4] B. Peemöller. *Normalization and Partial Evaluation of Functional Logic Programs*. Kiel Computer Science Series 2016/XY. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, Kiel University, 2016.

[5] S. L. Peyton Jones. *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.