UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Diplomarbeit

Elimination von Funktionen höherer Ordnung in Haskell–Programmen

von

Christian Schaller
Fürstenzeller Str. 41
94127 Neuburg

22. September 1998

Betreuung:

Dipl.-Inf. Christoph A. Herrmann

bei

Prof. Christian Lengauer
Lehrstuhl für Programmierung
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
94032 Passau

Meinem Vater.

Danksagung
An dieser Stelle möchte ich mich bei meinem Betreuer Christoph Herrmann für die Diskussionen und Anmerkungen während der Erstellung der vorliegenden Arbeit und natürlich für die exzellente Teeversorgung in den Projektbesprechungen bedanken. Besonderer Dank gilt auch Prof. Christian Lengauer für die konstruktive Kritik, die zum guten Gelingen dieser Arbeit erheblichen Anteil hatte. Nicht zu vergessen sind ebenfalls das Bibliothekspersonal des Lesesaals IV für die freundliche Unterstützung und meine Studienkollegen für die ausgedehnten Kaffeepausen. Großer Dank auch einem Mädchen namens Tina für die wunderbaren Ablenkungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Funktionale Programmierung erfreut sich einer immer größeren Beliebtheit bei der Erstellung von Programmen. Die Gründe dafür liegen vor allem in einer schnellen und kostengünstigen Entwicklung von Programmen unter der Berücksichtigung von Zuverlässigkeit, Fehlervermeidung und Wiederverwendbarkeit.

Eine der wichtigsten Eigenschaften von funktionalen Sprachen ist deren Unterstützung von Funktionen höherer Ordnung. Durch konsistente Verwendung von Funktionen höherer Ordnung können allgemeine Berechnungsmuster sehr knapp und präzise ausgedrückt werden. Dadurch wird es ermöglicht, wiederverwendbare Programme zu schreiben. Jedoch sind Funktionen höherer Ordnung schwer zu analysieren und zu optimieren. Sie bereiten außerdem Schwierigkeiten bei der Übersetzung funktionaler Sprachen in imperative Sprachen wie z.B. C.

In dieser Arbeit wird eine Source-to-Source–Transformation zur Elimination von Funktionen höherer Ordnung in funktionalen Sprachen vorgestellt, die ein funktionales Programm unter Erhaltung der Semantik in ein imperatives überführt.

Darüberhinaus wird die Entfunktionalisierung an die Haskell-ähnliche Sprache $\mathcal{HDC}$ angepaßt. Ein Compiler für $\mathcal{HDC}$ befindet sich am Lehrstuhl von Prof. Lengauer in Entwicklung.

Voraussetzung für das Verständnis dieser Diplomarbeit ist grundlegendes Wissen über funktionale Programmierung [Rea89, BW88, Tho96, Bir98]; wichtige Definitionen befinden sich in der Arbeit selbst.

Kapitel 1

Einführung

1.1 Motivation

In dem Papier „Parallelization of Divide-and-Conquer by Translation to Nested Loops“ [HL97] werden Programmtransformationen beschrieben, die ein mittels Divide-and-Conquer spezifiziertes Haskell–Programm [HJW92] in parallelen C–Code [KR78] übersetzen.

Divide-and-Conquer [Sed88] ist eine Vorgehensweise, bei der ein Problem in mehrere kleinere, unabhängige Teilprobleme aufgeteilt wird, die rekursiv gelöst werden, bis ein Basisfall auftritt. Die Lösungen der Teilprobleme werden anschließend kombiniert, um die Lösung des Ausgangsproblems zu erhalten.

Aufgrund der Wichtigkeit von Divide-and-Conquer existieren sogenannte algorithmische Skelette [DFH+93] als Basisblöcke, um dem Benutzer effiziente Implementierungen von Divide-and-Conquer zur Verfügung zu stellen.

Ein *Skelett* ist eine polymorphe Funktion höherer Ordnung, die ein algorithmisches Verfahren beschreibt. Eine Funktion wird *polymorph* genannt, wenn sie für eine Menge von Typen eine einheitliche Implementierung besitzt. Sie ist von *höherer Ordnung* (higher-order), falls sie Funktionen als Argumente besitzt oder als Ergebnis liefert. Ist eine Funktion nicht von höherer Ordnung, so wird sie allgemein als Funktion *erster Ordnung* bezeichnet. Ein Funktionsparameter heißt *Argumentfunktion*, wenn er eine Funktion ist.

In funktionalen Sprachen besitzen Funktionen denselben Status wie alle anderen Datenobjekte; sie können daher einer weiteren Funktion als Parameter übergeben werden. Darüberhinaus kann das Ergebnis einer Funktionsapplikation wieder eine Funktion sein.

Deswegen eignen sich funktionale Sprachen besonders gut zum Spezifizieren von parallelen Programmen durch Skelette. Die Skelette werden allgemein definiert und mit problemspezifischen Funktionen parametrisiert. Zusätzlich lassen sie sich leicht miteinander kombinieren.

Implementierungen von Skeletten sind in einer Funktionsbibliothek enthalten; sie können effizient parallel ausgeführt werden. Die allgemeinste Form eines Divide-and-Conquer–Skeletts ist definiert durch die Funktion `dc0`, die fünf Argumente benötigt.

Als Parameter erwartet `dc0` Funktionen mit den Typen

```
pred :: a -> Bool,
base :: a -> b,
divide :: a -> [a]
```

und

```
combine :: (a, [b]) -> b
```

(in Haskell–Syntax [HJW92]). Wendet man `dc0` mit diesen Funktionen auf ein Element mit Typ `a` an, so erhält man als Ergebnis einen Wert mit Typ `b`.

```
dc0 :: (a -> Bool) -> (a -> b) -> (a -> [a]) -> ((a, [b]) -> b)
       -> a -> b
dc0 pred base divide combine x = r x
  where
    r x = if pred x
          then base x
          else combine (x, map r (divide x))
```

Der Algorithmus `r` wird durch das Skelett `dc0` ausgedrückt, welches durch die folgenden Funktionen instanziiert wird: Das Prädikat `pred`, das den Basisfall erkennt, die Funktion `base`, die ausgeführt wird, wenn der Basisfall eintritt und die Funktion `divide`, die andernfalls benutzt wird, um das Problem in eine Liste von Teilproblemen zu teilen, die rekursiv gelöst werden. Die Funktion `combine` fügt schließlich die Teilergebnisse zusammen. Die Funktion `dc0` läßt sich noch spezialisieren zu weiteren Unterklassen von Divide-and-Conquer.

Zusätzlich zu den Eigenschaften von funktionalen Sprachen ermöglicht das Verbot von Seiteneffekten in Haskell, die Korrektheit durch Gleichheitsbeweise [O'D94, Bir98, Gün98] zu zeigen.

Eine wichtige Rolle bei der Spezifikation durch funktionale Sprachen spielt ebenfalls die sogenannte *Listenkomprehension* (list comprehension), deren Notation an die mathematische Mengenschreibweise angelehnt ist [BW88, S. 50ff]. Durch den Ausdruck

```
ys = [f x | x <- xs, x > 0]
```

wird die Liste `ys` mit Hilfe der Funktion `f` aus der Liste `xs` erzeugt. Dabei wird `f` auf jedes einzelne Element `x` aus `xs` angewandt, falls dieses größer als 0 ist; Elemente aus `xs`, die kleiner oder gleich 0 sind, werden verworfen. Den Ausdruck `x <- xs` bezeichnet man als *Generator,* `x > 0` als *Filter.*

Durch Listenkomprehensionen lassen sich datenparallele Berechnungen einfach darstellen. So kann im obigen Ausdruck jeder Prozessor die Funktion `f` auf einem Listenelement ausführen.

1.2 Problemstellung

Ziel ist es, in einer funktionalen Sprache (Haskell) spezifizierte Programme in eine imperative Sprache (C) zu überführen. Da jedoch in der Sprache C die Übergabe von Funktionen als Argumente konzeptuell nur unzureichend unterstützt wird, entsteht für die Verwaltung von Argumentfunktionen ein erhöhter Aufwand (siehe auch den nächsten Abschnitt). Um die Parallelisierung nicht zu beeinträchtigen, müssen sie daher eliminiert werden.

Beispiel:

Seien folgende Funktionsdefinitionen

```
map :: (a -> b) -> [a] -> [b]
map f ys
  = case ys of
    []     -> []
    (x:xs) -> f x : map f xs

inc :: Int -> Int
inc x = x + 1
```

mit dem Aufruf

```
map inc [1..10]
```

gegeben. Das Ergebnis des Aufrufs ist keine Funktion, jedoch ist `map` eine Funktion höherer Ordnung, da sie eine Argumentfunktion — die Funktion `inc` — beinhaltet, die eliminiert werden muß.

Diese Entfunktionalisierung muß für jedes funktionale Programm möglich sein; es darf keine Einschränkung an die Funktionsparameter geben. Auch Funktionen als Ergebnis von Applikationen, Argumentfunktionen in Rekursionen und in Konstruktoren sind zu eliminieren.

Das Resultat dieser Elimination soll ein gültiges und korrekt getyptes funktionales Programm sein. Dadurch wird es ermöglicht, eine große Anzahl von Transformationen (wie z.B. Code–Optimierungen), die nur für Funktionen erster Ordnung gelten, im Anschluß durchzuführen.

1.3 Behandlung von Funktionen höherer Ordnung

Eine der gebräuchlichsten Methoden zur Ausführung von funktionalen Programmen ist die Auswertung von Ausdrücken durch *Graphreduktion* [JL92, Kapitel 3].

Jeder Ausdruck wird durch einen Graphen dargestellt, der solange reduziert wird, bis er in *Normalform* (oder Vorstufe) ist, d.h. nicht mehr weiter reduzierbar ist (oder sinnvoll reduziert werden kann). Die sogenannte *G–Maschine* führt diese Graphreduktion durch.

Dazu wird ein Graph in einem Heap gespeichert; ein Stack mit Zeigern auf den Heap verwaltet Zwischenergebnisse der Reduktion, sowie die Argumente von Funktionen.

Bei der Übergabe von Argumentfunktionen wird für jede Funktion eine sogenannte *Funktionshülle* (Closure) [App92, AJ89] angelegt, eine Datenstruktur, die neben den freien Variablen der Funktion einen Zeiger auf den Code der Funktion enthält.

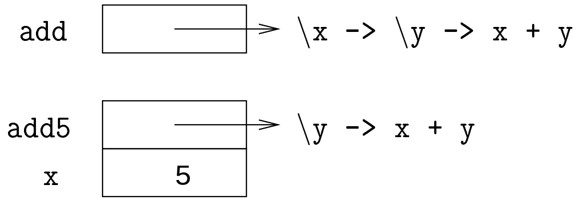

Abbildung 1.1: Funktionshüllen für `add` und `add5`

Beispiel:

Ein sehr einfaches Beispiel für den Umgang mit Funktionshüllen ist durch die Funktion

```
add = \x -> \y -> x + y
```

gegeben, die keine freien Variablen besitzt.
Wendet man `add` auf das Argument `5` an:

```
add5 = add 5,
```

so wird in der Funktionshülle von `add5` ein Zeiger auf die Funktion

```
\y -> x + y
```

mit der aktuellen Belegung `5` für die freie Variable `x` gespeichert.
Die Funktionshüllen für die beiden Funktionen sind in Abbildung 1.1 dargestellt.

Das zentrale Vorhandensein von Heap und Stack stellt jedoch einen Nachteil der G–Maschine dar. Es existieren zwar Modifikationen der G–Maschine zur parallelen Ausführung von Programmen [Jon92], jedoch wird der Parallelitätsgrad durch einen hohen Kommunikationsaufwand eingeschränkt. Eine statische Parallelisierung ist ebenfalls nicht gewährleistet.

1.3.1 Kodieren von Argumentfunktionen

Ein weiterer Ansatz zur Behandlung von Funktionen höherer Ordnung basiert auf dem Kodieren der Argumentfunktionen und der Einführung einer zusätzlichen Funktion, die kodierte Funktionen erst dann auswertet, wenn alle Argumente dafür vorhanden sind. Diese Methode wurde von Reynolds [Rey72] vorgestellt. Argumentfunktionen sind dann als Daten erster Ordnung kodiert und werden durch eine `apply`-Funktion ausgeführt.

Beispiel:

Aus der obigen Definition von `map` mit dem Aufruf

```
map inc [1..10]
```

entstehen folgende neue Funktionen:

```
map' f ys
  = case ys of
    []     -> []
    (x:xs) -> apply f x : map' f xs
```

und

```
apply f x
  = case f of
    "inc" -> inc x
```

mit dem ebenfalls modifizierten Aufruf

```
map' "inc" [1..10].
```

Die Argumentfunktion `inc` ist nun als Wert erster Ordnung

```
"inc" :: String
```

kodiert.

Diese Lösung stützt sich jedoch auf keinerlei Typinformation. Eine Argumentfunktion wird — egal welchen Typ sie besitzt — als String kodiert, und die `apply`-Funktion wird um die Dekodierung dieses Strings erweitert. Besitzt nun eine weitere Argumentfunktion einen anderen Typ als eine schon vorher kodierte, so stimmen die Typinformationen im Rumpf der `apply`-Funktion nicht mehr überein.

Beispiel:

Bei dem zusätzlichen Aufruf von

```
map int2bool [1..10]
```

mit

```
int2bool :: Int -> Bool
```

wird die `apply`-Funktion modifiziert zu

```
apply f x
  = case f of
    "inc"      -> inc x
    "int2bool" -> int2bool x.
```

Dies führt jedoch zu einem Typkonflikt, da `apply` wegen des ersten `case`–Zweiges den Typ

```
String -> Int -> Int
```

und wegen des zweiten `case`–Zweiges den Typ

```
String -> Int -> Bool
```

besitzt.

Typen müssen also auf jeden Fall berücksichtigt werden, wenn das Programm nach der Entfunktionalisierung wieder korrekt getypt sein soll.

Zusätzlich bereitet die Kodierung von Funktionen durch Strings Probleme, wenn sich Argumentfunktionen von Aufruf zu Aufruf ändern.

Beispiel:

Sei die Funktion:

```
sum f ys
  = case ys of
    []     -> f 0
    (x:xs) -> sum (\n -> x + f n) xs.
```

und der Ausdruck

```
sum id [1,2,3]
```

mit

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

gegeben.
Die Funktion `sum` läßt sich nicht einfach transformieren zu

```
sum' f ys
  = case ys of
    []     -> apply f 0
    (x:xs) -> sum' "(\n -> x + f n)" xs,
```

da im rekursiven Aufruf die Variablen `x` und `f` innerhalb des Strings auftreten.

Ähnliche Schwierigkeiten entstehen, wenn Ergebnisse von Funktionsapplikationen Funktionen sind.

1.3.2 Berücksichtigung von Typen

Aufbauend auf Reynolds Lösung [Rey72] wurde kürzlich eine Elimination von Funktionen höherer Ordnung vorgestellt, die die Idee von Reynolds erweitert und somit die oben genannten Probleme verhindert [BBH97].

Um Reynolds Beschränkungen zu beseitigen, wird die Elimination mittels Typinformation durchgeführt. Die Idee besteht darin, daß man verschiedene `apply`–Funktionen für unterschiedliche Typen generiert. Zusätzlich werden Kopien von Funktionen höherer Ordnung mit spezielleren Typen erzeugt. Diese Typen ergeben sich aus der jeweiligen Anwendungsstelle der Funktionen höherer Ordnung. Kopien von Funktionsdefinitionen mit spezielleren Typen werden als *Clones* bezeichnet.

Beispiel:

Will man wie oben sowohl den Aufruf

```
map inc [1..10]
```

als auch

```
map int2bool [1..10]
```

transformieren, so hat die Funktion `map` unter Berücksichtigung der Typen der beiden Funktionen `inc` und `int2bool` zwei spezielle Typen

(1) `(Int -> Int) -> [Int] -> [Int]`

(2) `(Int -> Bool) -> [Int] -> [Bool]`.

Zwei Kopien werden von `map` erstellt:

(1)
$$\texttt{map}_{\Psi_1}\ \texttt{f ys}$$
$$\quad \texttt{= case ys of}$$
$$\qquad \texttt{[]} \quad \texttt{-> []}$$
$$\qquad \texttt{(x:xs) -> apply}_{\texttt{Int -> Int}}\ \texttt{f x : map}_{\Psi_1}\ \texttt{f xs}$$

mit

$$\Psi_1 \texttt{ = (Int -> Int) -> [Int] -> [Int]}$$

und

(2)
$$\texttt{map}_{\Psi_2}\ \texttt{f ys}$$
$$\quad \texttt{= case ys of}$$
$$\qquad \texttt{[]} \quad \texttt{-> []}$$
$$\qquad \texttt{(x:xs) -> apply}_{\texttt{Int -> Bool}}\ \texttt{f x : map}_{\Psi_2}\ \texttt{f xs}$$

mit

$$\Psi_2 \texttt{ = (Int -> Bool) -> [Int] -> [Bool].}$$

Man beachte, daß die `apply`-Funktionen unterschiedlich sein müssen, da die Funktionen `inc` und `int2bool` verschiedene Typen besitzen.

Die Kodierung von Argumentfunktionen durch Strings erweist sich ebenfalls als nicht elegant, da nur konstante Argumentfunktionen kodiert werden können. Deswegen werden in diesem Fall Funktionen durch Konstruktoren kodiert:

(1) $\texttt{data T}_{\texttt{Int -> Int}} \texttt{ = C}^{\texttt{inc}}_{\texttt{Int -> Int}}$

(2) $\texttt{data T}_{\texttt{Int -> Bool}} \texttt{ = C}^{\texttt{int2bool}}_{\texttt{Int -> Bool}}$.

Schließlich fehlen noch die Definitionen der jeweiligen `apply`–Funktionen

(1)
$$\texttt{apply}_{\texttt{Int -> Int}}\ \texttt{c x}$$
$$\quad \texttt{= case c of}$$
$$\qquad \texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}} \texttt{ -> inc x}$$

(2)
$$\texttt{apply}_{\texttt{Int -> Bool}}\ \texttt{c x}$$
$$\quad \texttt{= case c of}$$
$$\qquad \texttt{C}^{\texttt{int2bool}}_{\texttt{Int -> Bool}} \texttt{ -> int2bool x.}$$

Die Aufrufe werden modifiziert zu

(1) $\texttt{map}_{\Psi_1}\ \texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}\ \texttt{[1..10]}$

und

(2) $\texttt{map}_{\Psi_2}$ $\texttt{C}^{\texttt{int2bool}}_{\texttt{Int -> Bool}}$ `[1..10]`.

Beispiel: nochmals die Funktion `sum`

Gegeben sei der Aufruf

```
sum id [1,2,3].
```

Es wird der Clone

```
sumΨ f ys
  = case ys of
    []     -> apply_{Int -> Int} f 0
    (x:xs) -> sumΨ (C^{(\n -> x + f n)}_{Int -> Int} x f) xs
```

mit

```
Ψ = (Int -> Int) -> Int -> Int
```

erzeugt. Die beiden Variablen `x` und `f` werden dem Konstruktor als Parameter übergeben; die Reihenfolge ist egal. Man beachte, daß die Variable `f` im rekursiven Aufruf von $\texttt{sum}_\Psi$ keine Funktion mehr repräsentiert, sondern einen Konstruktor. Dadurch ergibt sich die rekursive Typdefinition

```
data T_{Int -> Int}
  = C^{(\n -> x + f n)}_{Int -> Int} Int T_{Int -> Int}
  | C^{id}_{Int -> Int}
```

mit der `apply`–Funktion

```
apply_{Int -> Int} c y
  = case c of
    C^{id}_{Int -> Int}                 -> id y
    C^{(\n -> x + f n)}_{Int -> Int} x f -> x + apply_{Int -> Int} f y,
```

die ebenfalls rekursiv ist.
Der Aufruf wird transformiert zu

```
sumΨ C^{id}_{Int -> Int} [1..3].
```

Wurde vor der Transformation die Argumentfunktion im rekursiven Aufruf von `sum` um einen zusätzlichen λ-Ausdruck erweitert zu

```
(\n -> 3 + (\n -> 2 + (\n -> 1 + id n) n) n),
```

dem anschließend der Wert 0 übergeben wurde, so entsteht nach der Entfunktionalisierung der Baum von Konstruktoren in Abbildung 1.2, der durch die Funktion $\texttt{apply}_{\texttt{Int -> Int}}$ ausgewertet wird.

Wichtig ist bei der Entfunktionalisierung, daß die Typen der Clones hinreichend speziell sind, um die Typkonflikte aus Abschnitt 1.3.1 zu verhindern.

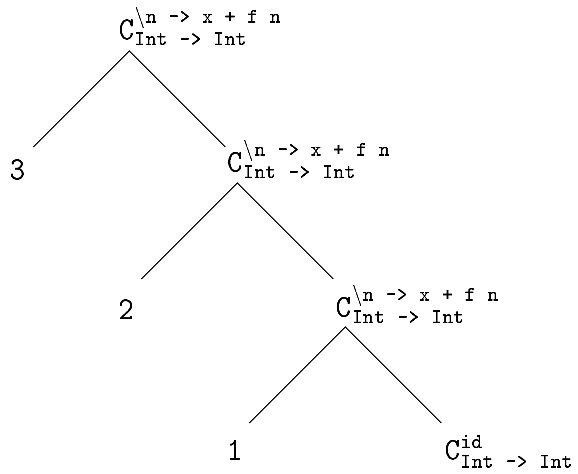

Abbildung 1.2: Der durch den Aufruf $\mathtt{sum}_\Psi$ $\mathtt{C}^{\mathtt{id}}_{\mathtt{Int\ ->\ Int}}$ `[1..3]` erzeugte Konstruktorbaum

Dies kann man z.B. dadurch erreichen, daß man die Transformation bei einem Ausdruck mit monomorphen Typen beginnt; alle darin vorkommenden polymorphen Funktionen besitzen in diesem Kontext speziellere Typen als in deren Definition. Es werden Kopien von den Funktionen erzeugt, die speziellere Typen besitzen. Dieser Vorgang wird rekursiv auf alle weiteren Ausdrücke fortgesetzt.

Eine auf diese Art transformierte Funktion ist isomorph zu einer Funktion erster Ordnung, da nun keine partiellen Funktionsapplikationen mehr existieren und deswegen Funktionsparameter als einziges Argument in Tupelform übergeben werden können (*Currying* [BW88, S. 12f]).

1.4 Die Sprache $\mathcal{HDC}$

In der Sprache $\mathcal{HDC}$ ($\mathcal{H}$igher-order–$\mathcal{D}$ivide-and-$\mathcal{C}$onquer) werden funktionale Programme durch eine Haskell-ähnliche Syntax spezifiziert. Dabei wird das Divide-and-Conquer Prinzip durch vordefinierte Skelette zur Generierung parallelen Codes benutzt. Ein am Lehrstuhl für Programmierung entwickelter Compiler übersetzt diese Programme in C–Code mit MPI–Aufrufen.

Eine große Rolle spielen in $\mathcal{HDC}$ auch die Listenkomprehensionen, die ein allgemeines Konzept für Parallelität darstellen. Deswegen ist z.B. die Funktion `map` in $\mathcal{HDC}$ im Gegensatz zu der Definition aus 1.2 durch folgende Listenkomprehension definiert:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs].
```

Im Unterschied zu Haskell sind in $\mathcal{HDC}$ die Programme strikt, es existieren keine Monaden und die Typklassen sind auf den Umgang mit den Typen `Int`, `Double` und `Bool` beschränkt. Ein vollständiges $\mathcal{HDC}$–Programm muß zusätzlich eine Hauptfunktion

```
parmain :: [Int] -> [Int]
```

besitzen, deren Name und Typ vorgegeben ist. Anhand dieses Typs läßt sich aus einem Programm mit polymorphen Funktionen ein für die Übersetzung in C–Code benötigtes monomorphes Programm ableiten.

Der Compiler ist aufgebaut wie in Abbildung 1.3 ersichtlich: Im Scanner/Parser wird ein Eingabeprogramm gelesen und in die interne Repräsentation überführt. Der Desugarer entfernt Konstrukte, die es dem Benutzer ermöglichen, Programme einfacher zu spezifizieren (sog. *syntaktischer Zucker*) und ersetzt sie durch semantisch äquivalente Ausdrücke. Anschließend wird eine Typprüfung durchgeführt, die den Typ jedes Ausdrucks berechnet. Die Entfunktionalisierung stellt die Grundlage dafür dar, daß ein $\mathcal{HDC}$–Programm statisch parallelisiert werden kann.

Im Anschluß wird der Code optimiert und eine Größeninferenz, gefolgt von der Berechnung eines parallelen Schedules durchgeführt. Vor der eigentlichen Codegenerierung wird abstrakter Code erzeugt, der sich leichter in verschiedene parallele Sprachen (C mit MPI [For93], High-Performance–Fortran [Sch97, KLS+94]) übersetzen läßt.

1.5 Übersicht

Im folgenden wird die Entfunktionalisierung nach [BBH97] eingeführt, und dort vorhandene Fehler werden korrigiert. Kapitel 2 zeigt die nötigen Voraussetzungen und führt Hilfsfunktionen ein, Kapitel 3 definiert die Transformationsregeln des ursprünglichen Verfahrens. Die Entfunktionalisierung wird in Kapitel 4 an die Bedürfnisse des Projektes $\mathcal{HDC}$ angepaßt. Ein Vergleich der beiden Transformationen, sowie eine Untersuchung der Laufzeit entfunktionalisierter Programme in Kapitel 5 bewerten die in dieser Arbeit vorgestellten Methoden. Abschließend werden in Kapitel 6 Modifikationen vorgestellt, die das Laufzeitverhalten entfunktionalisierter Programme verbessern. Eine Diskussion der Ergebnisse beendet diese Arbeit.

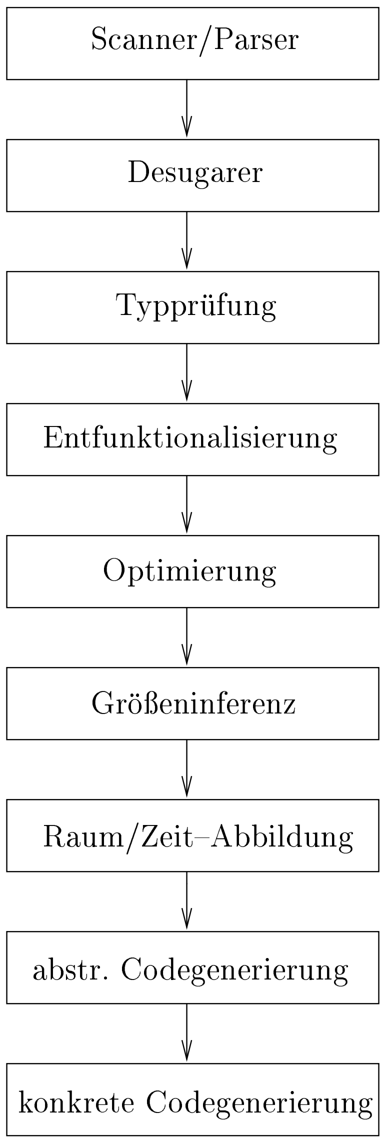

Abbildung 1.3: Die Phasen des Compilers

Kapitel 2

Grundlagen

In diesem Kapitel werden Vorbereitungen für die im Anschluß vorgestellte Elimination von Funktionen höherer Ordnung getroffen. Dazu werden in Abschnitt 2.1 Voraussetzungen an das Eingabeprogramm diskutiert, die erfüllt sein müssen, damit die Entfunktionalisierung erfolgreich durchgeführt werden kann. Die für die Transformation benötigten internen Datentypen und Hilfsfunktionen werden in den Abschnitten 2.2 und 2.3 vorgestellt.

2.1 Voraussetzungen

Die Eingabesprache für die Entfunktionalisierung ist angelehnt an Haskell, jedoch ohne Typklassen. Sie kann aber jede beliebige funktionale Sprache mit algebraischen Datentypen, `case`–, `if-then-else`–Ausdrücken und Mustervergleichen sein.

Zulässige Programme dürfen keine `let`–Ausdrücke, insbesondere keine lokalen Funktionen beinhalten. Diese Bedingung stellt jedoch keine Einschränkung der Eingabesprache dar, da lokale Funktionen und `let`–Ausdrücke durch sogenanntes λ–*Lifting* [Joh85] aufgelöst werden können. Diese und ähnliche Vorverarbeitungen finden im $\mathcal{HDC}$–Compiler während der Desugarer–Phase statt (vgl. Abbildung 1.3).

Listenkomprehensionen stellen ein wichtiges Mittel zum Ausdrücken von parallelen Vorgängen dar und werden daher nicht durch einfachere Ausdrücke ersetzt.

Für die Elimination von Funktionen höherer Ordnung ist es ferner notwendig, geschlossene Programme zu transformieren. Ein Programm heißt *geschlossen*, wenn alle darin auftretenden Funktionen definiert sind.

Wie in Abschnitt 1.3.2 schon bemerkt wurde, sind Typinformationen für die Entfunktionalisierung von großer Bedeutung. Aus diesem Grunde ist es unerläßlich, daß jeder Ausdruck des Eingabeprogramms den richtigen Typ besitzt.

2.2 Datentypen

Programme werden — hier zur Präsentation der Elimination von Funktionen höherer Ordnung vereinfacht — in Definitionen algebraischer Datentypen und

Funktionen aufgeteilt. Die in den Algorithmen verwendeten internen Datentypen sind

- `Data` zur Repräsentation von algebraischen Datentypen,
- `Fun` zur Repräsentation von Funktionsdefinitionen,
- `Expr` zur Repräsentation von Ausdrücken und
- `Type` zur Repräsentation von Typen.

Diese verwendeten Datentypen sind baumartig strukturiert. Eine Funktion wird intern als Paarung des Funktionsnamens und des definierenden λ–Ausdrucks dargestellt:

```
type Fun = (String, Expr).
```

2.3 Hilfsfunktionen

Die im folgenden vorgestellten Funktionen zur Entfunktionalisierung sind in einer Haskell-ähnlichen Syntax angegeben. Zur Unterscheidung von Haskell–Programmen werden Schlüsselwörter unterstrichen dargestellt und sind ins Deutsche übersetzt. Ebenfalls wird in Anlehnung an `->` aus `case`–Ausdrücken $\Longrightarrow$ in Fallunterscheidungen verwendet.

Die Hilfsfunktionen `countArgs` und `isFun` in Abbildung 2.1 dienen zum Erkennen von Argumentfunktionen. Ein Ausdruck heißt *funktionsenthaltend*, wenn er vom Typ her eine Funktion ist oder eine Datenstruktur, die eine Funktion enthält. Die Prädikatfunktion `containsFun` in Abbildung 2.2 zeigt anhand des Typen eines Ausdrucks an, ob er funktionsenthaltend ist.

Die Prädikatfunktionen `isFun` und `containsFun` unterscheiden sich in der Behandlung von algebraischen Datentypen: im Gegensatz zu `isFun` werden durch `containsFun` auch Funktionen erkannt, die in einem Konstruktorausdruck auftreten, wie im folgenden Beispiel ersichtlich ist.

Beispiel:

In dem Ausdruck

```
map inc xs
```

mit

```
inc :: Int -> Int
```

ergibt `isFun (Int -> Int)` den Wert `True`, d.h. `inc` ist eine Argumentfunktion. Im Gegensatz dazu ergibt im Ausdruck

```
mapf [inc] x
```

mit

```
mapf :: [a -> b] -> a -> [b]
```

```
countArgs :: Type -> Int
countArgs type
  = falls type von der Form
    (a -> b)  ⟹ 1 + countArgs b
    andernfalls  ⟹ 0

isFun :: Type -> Bool
isFun type = countArgs type > 0
```

Abbildung 2.1: Definition der Funktionen `countArgs` und `isFun`

```
containsFun :: Type -> Bool
containsFun type
  = falls type von der Form
    (a -> b)  ⟹ True
    T t1...tn  ⟹
      (containsFun t1 ∨...∨ containsFun tn)
        ∨ (∃ Konstruktor C aus der Definition von T mit
          C :: c1 ->...-> cm -> T t1...tn
          und containsFun c1 ∨...∨ containsFun cm)
    andernfalls  ⟹ False
```

Abbildung 2.2: Definition der Prädikatfunktion `isFun`

und

```
[inc] :: [Int -> Int]
```

`isFun [Int -> Int]` den Wert `False`, da die Funktion `inc` in einer Liste an `mapf` übergeben wird. Die Funktion `containsFun` berücksichtigt auch diesen Fall; `containsFun [Int -> Int]` ergibt den Wert `True`, d.h. der Funktionsparameter `[inc]` enthält eine Funktion, die ebenfalls eliminiert werden muß.

Während der Transformation muß ein Ausdruck mit zusätzlicher Typinformation versehen werden; diese fügt die Funktion `mark` aus Abbildung 2.3 dem Ausdruck textuell hinzu. Nach erfolgreicher Transformation werden die Marken mit der Funktion `unmark` (Abbildung 2.4) wieder entfernt.

```
mark :: (Expr, Type) -> Expr
mark (e, t) = <e>_t
```

Abbildung 2.3: Definition von `mark`

```
unmark :: Expr -> Expr
unmark expr
    = falls expr von der Form
      <v>_t  ==> v
      <f e_1 ... e_n>_t -> f (unmark e_1) ... (unmark e_n),
          falls f Operator, Funktionssymbol, Funktionsvariable
            oder Konstruktor
      <case e of {p_1 -> e_1; ...; p_n -> e_n}>_t
        ==> case (unmark e) of
            {p_1 -> (unmark e_1); ...; p_n -> (unmark e_n)}
      <[e | gf_1, ..., gf_n]>_t
        ==> [(unmark e) | (unmark gf_1), ..., (unmark gf_n)]
```

Abbildung 2.4: Definition von `unmark`

Kapitel 3

Entfunktionalisierung

Die Elimination von Funktionen höherer Ordnung in [BBH97] besteht aus den folgenden Transformationsregeln, die auf den Quellcode angewendet werden.

Die Typspezialisierung (Abschnitt 3.1) kopiert Funktionsdefinitionen und versieht sie mit spezielleren Typen, die sich aus der Anwendungsstelle der jeweiligen Funktionen ergeben.

Bei der Kodierung (Abschnitt 3.2) werden Argumentfunktionen durch Konstruktoren ersetzt und Datentypen kreiert, zu denen diese Konstruktoren gehören. Zusätzlich werden Funktionen zur Dekodierung definiert, die die durch Konstruktoren repräsentierten Funktionen aufrufen.

Da nach der Kodierung keine Funktionen, sondern Konstruktoren als Argumente übergeben werden, repräsentieren Variablen nun kodierte Funktionen. Existierten vor der Transformation Applikationen von Funktionsvariablen, so müssen die Konstruktoren nun zur Ausführung der Funktionen dekodiert werden (Abschnitt 3.3).

Neben diesen Transformationsregeln existieren noch zwei weitere zur Aktualisierung von Marken (Abschnitte 3.4 und 3.5).

Eine Regel überführt ein Tupel, bestehend aus einem Ausdruck, der untersucht werden soll, allen benutzerdefinierten Datentypen und allen Funktionsdefinitionen, in ein neues Tupel, das den modifizierten Ausdruck, modifizierte algebraische Datentypen und modifizierte Funktionsdefinitionen enthält. Diese Transformationsregeln werden solange auf alle Ausdrücke angewandt, bis keine Änderung mehr vorgenommen werden kann.

Im Anschluß an diese Iteration finden weitere Transformationen statt (Abschnitte 3.6 und 3.7). Man erhält dadurch unabhängig von der Ausführungsreihenfolge der Regeln ein entfunktionalisiertes Programm, das sich in ein C–Programm übersetzen läßt.

Eine Vorverarbeitungsphase bereitet den Code für die Elimination von Funktionen höherer Ordnung vor. Abbildung 3.1 gibt einen Überblick über die in diesem Kapitel vorgestellte Entfunktionalisierung.

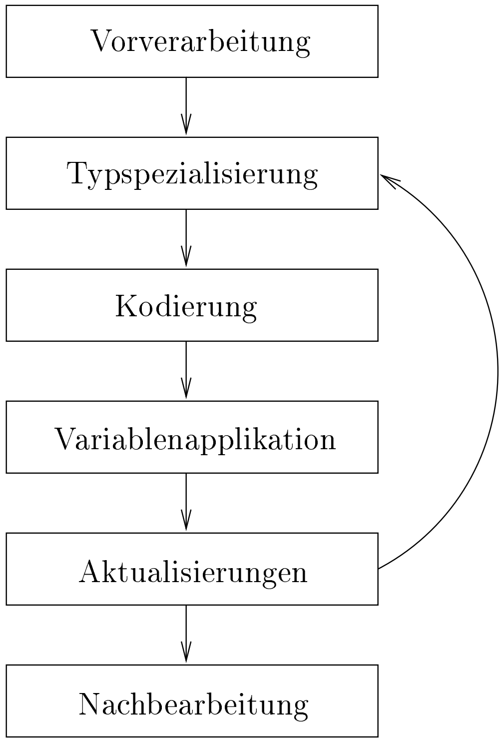

Abbildung 3.1: Die Phasen der Entfunktionalisierung

3.1 Typspezialisierung

Ziel der Typspezialisierung ist es, Kopien von Funktionen höherer Ordnung zu erzeugen, die an der Position funktionsenthaltender Argumente monomorphe Typen besitzen. Da jetzt für funktionsenthaltende Argumente mit unterschiedlichen Typen verschiedene Kopien von Funktionen existieren, treten keine Typkonflikte mehr auf (vgl. Abschnitt 1.3.1).

Beispiel:

Sei der Aufruf

```
map inc [1..10]
```

mit

```
map :: (a -> b) -> [a] -> [b],

[1..10] :: [Int]
```

und

```
inc :: Int -> Int
```

gegeben.

Durch Unifikation der Typen aus der Definition von `map` mit den aktuellen Parametern ergibt sich die Substitution `[Int/a, Int/b]` (Notation aus [Bro92]) und daraus der im Kontext verwendete spezielle Typ von `map`

`(Int -> Int) -> [Int] -> [Int] =` Ψ.

Anhand dieser Information wird der Clone

$\texttt{map}_{\Psi}$ :: `(Int -> Int) -> [Int] -> [Int]`
$\texttt{map}_{\Psi}$ `f xs = [<f>`$_{\texttt{Int -> Int}}$ `x | x <- xs]`

hinzugefügt, in dem die Typvariablen `a` und `b` durch `Int` ersetzt werden. Zusätzlich werden funktionsenthaltende Variablen mit den im Kontext verwendeten, evtl. spezielleren Typen markiert.
Der Aufruf wird modifiziert zu

$\texttt{map}_{\Psi}$ `inc [1..10]`.

Voraussetzungen für die Typspezialisierung

Im folgenden werden die Bedingungen vorgestellt, die gelten müssen, damit eine Funktion spezialisiert werden kann. Sei dazu der Aufruf

$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n$

mit

$(\forall j : 1 \leq j \leq n : \texttt{e}_j :: \Pi_j)$
$\texttt{f} :: \Pi$
$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n :: \Omega$

gegeben. Dieser Ausdruck läßt sich spezialisieren, wenn folgende Bedingungen erfüllt sind:

(i) `countArgs` Ω `== 0`

(ii) (`f` ist Funktionssymbol) $\wedge$ (`f` ist kein Clone)

(iii) $(\exists j : 1 \leq j \leq n : (\texttt{containsFun}\ \Pi_j))$

(iv) alle funktionsenthaltenden Variablen in funktionsenthaltenden Argumenten sind markiert

Dabei stellt Bedingung (i) sicher, daß nur *gesättigte* Ausdrücke transformiert werden, d.h. Resultate sind erster Ordnung. Die Behandlung von Applikationen, deren Ergebnisse Funktionen sind, wird durch eine weitere Transformationsregel sichergestellt.

Bedingung (ii) verhindert, daß Funktionsvariablen transformiert werden, da nur definierte Funktionen spezialisiert werden können. Clones sind durch erfolgreiche Anwendung der Typspezialisierung entstanden und werden deswegen nicht mehr bearbeitet.

Die Funktion `f` wird nur dann spezialisiert, wenn sie funktionsenthaltende Argumente besitzt (Bedingung (iii)).

Ein Clone einer Funktion darf erst dann erzeugt werden, wenn die Typinformationen ausreichen, damit in der später erzeugten `apply`–Funktion kein Typkonflikt auftritt. Dies wird durch Bedingung (iv) berücksichtigt.

Beispiel:

Die Funktion

```
foo :: (a -> b) -> [a] -> [b]
foo f xs = map (id f) xs
```

beinhaltet den Aufruf

`map (id f) xs`,

der alle Bedingungen außer (iv) erfüllt. Die Transformation darf noch nicht angewendet werden, da zu wenig Typinformation vorhanden ist; `map` kann noch nicht genügend spezialisiert werden.
Würde trotzdem der Aufruf in `foo` transformiert werden zu

$\mathtt{map}_{\Psi}$ `(id f) xs`

mit

$\Psi =$ `(a -> b) -> [a] -> [b]`,

so entstünde eine `apply`–Funktion für die Argumentfunktion `(id f)`, die alle Funktionen vom Typ `a -> b` aufrufen würde.
Bei Verwendung der Funktionen

`int2bool :: Int -> Bool`

und

`bool2int :: Bool -> Int`

für `f` entstünde erneut das Typproblem aus Abschnitt 1.3.1.
Nach Regel (iv) kann `map` erst geklont werden, wenn `foo` spezialisiert ist und dadurch `f` markiert ist.

Man beachte, daß bei der Typspezialisierung die Typen der Argumente mit dem Prädikat `containsFun` anstelle von `isFun` getestet werden, um auch den Fall abzudecken, daß Funktionen in Datenstrukturen auftreten.

Definition der Transformationsregel `funSpec`

Sind alle obigen Bedingungen erfüllt und gilt zusätzlich `countArgs` $\Pi == n$, d.h. die Funktion `f` besitzt genausoviele Parameter wie in der Definition angegeben, so wird der Aufruf

$\mathtt{f}\ \mathtt{e}_1 \ldots \mathtt{e}_n$

mit der Definition von `f`

$\mathtt{f}\ \mathtt{x}_1 \ldots \mathtt{x}_n\ \mathtt{=}\ \mathtt{D}$

transformiert zu

$$\texttt{f}_{\sigma\Pi}\ \texttt{e}_1 \ldots \texttt{e}_n$$

mit

$$\sigma\Pi\ \texttt{=}\ \Pi_1\ \texttt{->}\ldots\texttt{->}\ \Pi_n\ \texttt{->}\ \Omega$$

und der Clone

$$\begin{aligned}&\texttt{f}_{\sigma\Pi}\ \texttt{x}_1 \ldots \texttt{x}_n\\ &\quad \texttt{= D [<}\texttt{x}_{i_1}\texttt{>}_{\Pi_{i_1}}\texttt{/}\texttt{x}_{i_1}\texttt{,}\ldots\texttt{,<}\texttt{x}_{i_k}\texttt{>}_{\Pi_{i_k}}\texttt{/}\texttt{x}_{i_k}\texttt{]}\end{aligned}$$

wird dem Programm hinzugefügt mit

i_j Indizes der funktionsenthaltenden Argumente

und

$1 \leq j \leq k$, k Anzahl der funktionsenthaltenden Argumente.

Ist jedoch `countArgs` Π `<` n, d.h. `f` wird mit mehr aktuellen Parametern verwendet als sie mit formalen Parametern definiert ist, muß zusätzlich zur Spezialisierung eine η-*Erweiterung* ihrer Definition stattfinden. Die η-*Erweiterung* ist die Inverse zur η-*Reduktion* [Mic89, Geo84, CD96], bei der λ-Ausdrücke der Form `(\n -> expr n)` reduziert werden zu `expr`.

Beispiel:

Zur Darstellung der η-Erweiterung sei die einfachste Situation mit dem Ausdruck

```
id id 7
```

und der Definition

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

gegeben. Die Funktion `id` besitzt im ersten Fall die Argumentfunktion `id` als Parameter. Der aus dieser Applikation entstehenden Funktion wird der Integer–Wert 7 übergeben. Zur besseren Unterscheidung der beiden Funktionen wird `id'` für die erste Identität benutzt:

```
id' id 7.
```

Der spezielle Typ von `id'` berechnet sich zu

```
id' :: (Int -> Int) -> Int -> Int.
```

Die Funktion `id'` ist also eine Funktion mit zwei Parametern, d.h. `id'` muß um eine Variable erweitert werden, bevor sie spezialisiert werden kann zu

$$\begin{aligned}&\texttt{id'}_\Psi \texttt{ :: (Int -> Int) -> Int -> Int}\\ &\texttt{id'}_\Psi \texttt{ x eta = x eta}\end{aligned}$$

```
(◇) :: Expr -> Expr -> Expr
e ◇ v
  = falls e von der Form
    (case c of {p1 -> e1; ...; pn -> en})
      ⟹ (case c of {p1 -> (e1 ◇ v); ...; pn -> (en ◇ v)})
    (if c then e1 else e2)
      ⟹ (if c then (e1 ◇ v) else (e2 ◇ v))
    andernfalls -> (\v -> e v)
```

Abbildung 3.2: Definition der Funktion (◇)

mit Ψ = `(Int -> Int) -> Int -> Int`.
Man erhält so den modifizierten Aufruf

$$\texttt{id'}_{\Psi}\ \texttt{id 7}.$$

Bei dem Hinzufügen von Variablen ist bei `case`–Ausdrücken zu beachten, daß die Variable nicht einfach angehängt werden darf, sondern in jeden einzelnen `case`–Zweig propagiert werden muß; analoges gilt auch für `if-then-else`–Ausdrücke. Dies wurde in [BBH97] nicht berücksichtigt, da die Inputsprache ohne `if-then-else`–Ausdrücke definiert wurde. Die Funktion (◇) in Abbildung 3.2 fügt zu einem Ausdruck eine Variable hinzu, unter Berücksichtigung von `case`– und `if-then-else`–Ausdrücken. Der vollständige Algorithmus für die Typspezialisierung ist in Abbildung 3.3 wiedergegeben.

Der in [BBH97, S. 29–30] vorgestellte Algorithmus entfernt zusätzlich alle Markierungen in den modifizierten Funktionsapplikationen, was aber zu unvollständig transformierten Programmen führt.

Beispiel:

Sei folgender Clone gegeben:

$$\texttt{foo}_{\Psi}\ \texttt{f g x xs = map (<f>}_{\Pi_1}\ \texttt{<g>}_{\Pi_2}\ \texttt{x) xs}.$$

Der Aufruf von `map` läßt sich spezialisieren zu

$$\texttt{foo}_{\Psi}\ \texttt{f g x xs = map}_{\sigma\Pi}\ \texttt{(<f>}_{\Pi_1}\ \texttt{<g>}_{\Pi_2}\ \texttt{x) xs}.$$

Würde man nun, wie in den Transformationsregeln zur Funktionsspezialisierung aus [BBH97], die Markierungen löschen, könnten die nachfolgenden Regeln nicht angewendet werden, weil sie eine mit (iv) vergleichbare Bedingung erfüllen müssen. Da nur während der Funktionsspezialisierung Markierungen eingeführt werden und diese wegen Bedingung (ii) nicht mehr auf `map` angewendet werden kann, würden `f` und `g` unmarkiert bleiben und die Transformation wäre unvollständig.

Aus diesem Grund wurde in dem hier vorgestellten Algorithmus auf das Entfernen von Marken verzichtet.

```
funSpec :: (Expr, [Data], [Fun]) -> (Expr, [Data], [Fun])
funSpec (e, ds, fs)
  = wenn e Applikation der Form:  f $e_1 \ldots e_n$
      $\wedge$ f ist Funktionssymbol $\wedge$ f ist kein Clone
      $\wedge$ countArgs $\Omega$ == 0
      $\wedge$ ($\exists j: 1 \leq j \leq n$: containsFun $\Pi_j$)
      $\wedge$ alle funktionsenthaltenden Variablen in
        funktionsenthaltenden Argumenten sind markiert
    dann
      sei
        berechne speziellen Typ $\sigma\Pi$:
        $\sigma\Pi$ = $\Pi_1$ ->...-> $\Pi_n$ -> $\Omega$
        ändere e zu:
        e' = $f_{\sigma\Pi}$ $e_1 \ldots e_n$
        füge, wenn nötig, neue Variablen zur Definition von f:
        D' = foldl ($\diamond$) D [$v_1, \ldots, v_{n\text{-}p}$]
        erzeuge Clone:
        f' = ($f_{\sigma\Pi}$,\$x_1$ ->...-> \$x_p$ -> \$v_1$ ->...-> $v_{n\text{-}p}$
              -> (D' [<$x_{i_1}$>$_{\Pi_{i_1}}$/$x_{i_1}$, ..., <$x_{i_k}$>$_{\Pi_{i_k}}$/$x_{i_k}$,
                     <$v_{l_1}$>$_{\Pi_{l_1}}$/$v_{l_1}$, ..., <$v_{l_m}$>$_{\Pi_{l_m}}$/$v_{l_m}$])
        mit $i_\xi$, $l_\zeta$ Indizes der funktionsenthaltenden Argumente,
        $1 \leq \xi \leq k, 1 \leq \zeta \leq m$,
        $k$ und $m$ Anzahl der funktionsenthaltenden Argumente
      in (e', ds, f' : fs)
    sonst (e, ds, fs)
  wobei
    f :: $\Pi$
    p = countArgs $\Pi$
    f $e_1 \ldots e_n$ :: $\Omega$
    f ist definiert als (f,\$x_1$ ->...-> \$x_p$ -> D)
    $e_i$ :: $\Pi_i$,  $1 \leq i \leq n$
```

Abbildung 3.3: Die Typspezialisierungsregel `funSpec`

3.2 Kodierung von Argumentfunktionen

Nach dem Klonen von Funktionen müssen Argumentfunktionen eliminiert werden. Dazu werden sie durch Konstruktoren kodiert und diese treten an die Stelle der Argumente. Um den Zusammenhang von Funktion und Konstruktor herzustellen, müssen zusätzliche top-level Funktionen (sogenannte `apply`–Funktionen) hinzugefügt werden, die je nach Konstruktor die betreffende Funktion ausführen.

Beispiel:

Sei der durch die Typspezialisierung entstandene Funktionsaufruf

$$\texttt{map}_{\Psi}\ \texttt{inc [1..10]}$$

gegeben.
Die Argumentfunktion `inc` wird durch den Konstruktor $\texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}$ ersetzt

$$\texttt{map}_{\Psi}\ \texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}\ \texttt{[1..10]}.$$

Zusätzlich wird ein neuer Datentyp erzeugt, zu dem dieser Konstruktor gehört:

$$\texttt{data T}_{\texttt{Int -> Int}} = \texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}.$$

Zur Übersicht steht im Superskript des Konstruktors die Funktion, die kodiert wurde und im Subskript der evtl. speziellere Typ dieser Funktion.
Die hinzugefügte `apply`–Funktion

$$\begin{array}{l}\texttt{apply}_{\texttt{Int -> Int}}\ \texttt{f x}\\ \quad \texttt{= case f of}\\ \qquad \texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}\ \texttt{-> inc x}\end{array}$$

dient zur Ausführung der durch den Konstruktor $\texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}$ kodierten Funktion `inc`, sowie aller weiterer kodierter Funktionen vom Typ `Int -> Int`.

Voraussetzungen für die Kodierung

Sei der Ausdruck

$$\texttt{f e}_1 \ldots \texttt{e}_n$$

mit

$$\begin{array}{l}(\forall j: 1 \leq j \leq n: \texttt{e}_j :: \Pi_j)\\ \texttt{f} :: \Pi\\ \texttt{f e}_1 \ldots \texttt{e}_n :: \Omega\end{array}$$

gegeben. Damit die Kodierung von Funktionen in Konstruktoren durchgeführt werden kann, müssen folgende Bedingungen erfüllt sein:

(i) $\texttt{countArgs}\ \Omega\ \texttt{==}\ 0$

(ii) $(\exists j: 1 \leq j \leq n: (\texttt{isFun}\ \Pi_j) \wedge (\texttt{e}_j$ ist keine Variable$)$
$\wedge$ (alle funktionsenthaltenden Variablen in $\texttt{e}_j$ sind markiert$))$

(iii) `f` ist ein Clone, eine `apply`-Funktion oder ein Konstruktor

Dabei stellt Bedingung (ii) sicher, daß keine Variablen kodiert werden, sondern nur definierte Argumentfunktionen; Variablen repräsentieren lediglich Werte.

Damit die Semantik des Ausgangsprogramms durch die Elimination von Funktionen höherer Ordnung nicht verändert wird, muß Bedingung (iii) erfüllt sein. Dadurch wird ein Fehler in der Definition der Transformationsregel `encode` in [BBH97, S. 35] behoben, in dem `f` ein Funktionssymbol, eine Funktionsvariable oder ein Konstruktor sein kann.

Beispiel:

Sei folgende top-level Funktion gegeben:

```
bar f xs = f inc xs.
```

Diese Funktion darf nur semantik- und typerhaltend transformiert werden, da es sich hierbei um keinen Clone handelt.
Bei Vernachlässigung von Bedingung (iii) würde diese Definition jedoch transformiert werden zu

$$\texttt{bar f xs = f } \texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}} \texttt{ xs.}$$

Die Funktion `bar` hätte nicht mehr den ursprünglichen Typ und könnte deshalb im Kontext nicht mehr verwendet werden.

Definition der Transformationsregel `encode`

Seien alle Bedingungen für die Anwendung der Transformationsregel erfüllt. Der Ausdruck

$$\texttt{f } \texttt{e}_1 \ldots \texttt{e}_{j-1} \ \texttt{e}_j \ \texttt{e}_{j+1} \ldots \texttt{e}_n$$

mit der Argumentfunktion $\texttt{e}_j$ wird transformiert zu

$$\texttt{f } \texttt{e}_1 \ldots \texttt{e}_{j-1} \ (\texttt{C}^{\texttt{e}_j}_{\Pi_j} \ \texttt{v}_1 \ldots \texttt{v}_k) \ \texttt{e}_{j+1} \ldots \texttt{e}_n,$$

$\texttt{v}_1,\ldots,\texttt{v}_k$ sind freie Variablen in $\texttt{e}_j$.
Der Datentyp $\texttt{T}_{\Pi_j}$ wird erzeugt, bzw. falls dieser schon existiert, aktualisiert.

$$\begin{array}{l}\texttt{data } \texttt{T}_{\Pi_j} \ \alpha_1 \ldots \alpha_m \\ \quad \texttt{= } \texttt{C}^{\texttt{e}_j}_{\Pi_j} \ \Phi_1 \ldots \Phi_k\end{array}$$

mit

$$\Phi_i = \begin{cases} \Psi_i, & \text{falls } \texttt{v}_i \text{ mit } \Psi_i \text{ markiert ist} \\ \beta_i, & \text{sonst} \end{cases} , \qquad 1 \leq i \leq k$$

und $\alpha_1, \ldots, \alpha_m$ Typvariablen in $\Phi_1, \ldots, \Phi_k$ und β_i neue Typvariablen.

Korrektes Hinzufügen eines Konstruktors wird durch die Funktion $(\triangle)$ aus Abbildung 3.4 gewährleistet. Wichtig ist hierbei die Funktion μ, die die Typvariablen des algebraischen Datentyps (`Data`) nach Hinzufügen eines neuen Konstruktors aktualisiert.

Zur Dekodierung der Konstruktoren wird eine `apply`–Funktion erzeugt, bzw. ein neuer `case`–Zweig einer schon bestehenden `apply`–Funktion hinzugefügt:

$(\triangle)$ :: `Data -> [Data] -> [Data]`
`d` $\triangle$ `[] = [d]`
`(data` T_Π `=` C^e_Π $\Phi_1 \ldots \Phi_n$`)` $\triangle$ `((data` T_Π $\alpha_1 \ldots \alpha_m$ `= cs) : ds)`
  `= (`μ `(data` T_Π `= cs |` C^e_Π $\Phi_1 \ldots \Phi_n$`) : ds)`
`d'` $\triangle$ `(d : ds)`
  `= d : (d'` $\triangle$ `ds)`

μ :: `Data -> Data`
μ `(data T =` c_1 $|\ldots|$ c_n`)`
  `= data T` $\alpha_1 \ldots \alpha_k$ `=` c_1 $|\ldots|$ c_n
  <u>`wobei`</u> $\alpha_1, \ldots, \alpha_k$ `Typvariablen in` $c_1, \ldots, c_n$

Abbildung 3.4: Definition von $(\triangle)$ und μ

$$\begin{array}{l} \texttt{apply}_{\Pi_j}\ \texttt{c}\ \texttt{x}_1 \ldots \texttt{x}_{\texttt{countArgs}\ \Pi_j} \\ \quad \texttt{= case c of} \\ \qquad \texttt{C}^{\texttt{e}_j}_{\Pi_j}\ \texttt{v}_1 \ldots \texttt{v}_k\ \texttt{->}\ \texttt{foldl}\ (\diamond)\ \texttt{e}_j\ [\texttt{x}_1, \ldots, \texttt{x}_{\texttt{countArgs}\ \Pi_j}]. \end{array}$$

Zu beachten ist, daß die $\texttt{x}_i$ ($1 \leq i \leq \texttt{countArgs}\ \Pi_j$) dem Ausdruck $\texttt{e}_j$ mit der Funktion $(\diamond)$ angefügt werden müssen, da e_j z.B. ein `case`–Ausdruck sein kann. Dies wurde bei der Definition der Regel `encode` in [BBH97, S. 35] ebenso wenig berücksichtigt, wie das Markieren der $\texttt{x}_i$, wenn sie funktionsenthaltend sind.

Beispiel:

Die Funktion `map` werde einer Funktion als Parameter übergeben. Sie wird durch den Konstruktor $\texttt{C}^{\texttt{map}}_\Psi$ kodiert. Es entsteht folgende `apply`–Funktion:

$$\begin{array}{l} \texttt{apply}_\Psi\ \texttt{c}\ \texttt{x}_1\ \texttt{x}_2 \\ \quad \texttt{= case c of} \\ \qquad \texttt{C}^{\texttt{map}}_\Psi\ \texttt{->}\ \texttt{map}\ \texttt{x}_1\ \texttt{x}_2. \end{array}$$

Der Funktionsaufruf $\texttt{map}\ \texttt{x}_1\ \texttt{x}_2$ muß durch die Regel `funSpec` noch spezialisiert werden. Dazu sollte aber das funktionsenthaltende Argument $\texttt{x}_1$ markiert sein, damit die Bedingung (iv) in Abschnitt 3.1 erfüllt ist.

Das korrekte Hinzufügen von `apply`–Funktionen bzw. das Aktualisieren von `case`–Zweigen in `apply`–Funktionen wird durch die Funktion (∇) aus Abbildung 3.5 gewährleistet. Die vollständige Definition der Transformationsregel `encode` ist in Abbildung 3.6 dargestellt.

Beispiel:

Der Aufruf

$$\texttt{map}_\Psi\ \texttt{(add x) xs}$$

```
(∇) :: Fun -> [Fun] -> [Fun]
f ∇ [] = [f]
(apply_Π c v_1...v_n = case c of p' -> e')
  ∇ ((apply_Π c v_1...v_n = case c of pes) : as)
  = (apply_Π c v_1...v_n = case c of pes | p' -> e') : as
a' ∇ (a : as)
  = a : (a' ∇ as)
```

Abbildung 3.5: Definition der Funktion (∇)

```
encode :: (Expr, [Data], [Fun]) -> (Expr, [Data], [Fun])
encode (e, ds, fs)
  = wenn e Applikation der Form:  f e_1...e_n
      ∧ f ist entweder Clone, apply-Funktion oder Konstruktor
      ∧ countArgs Ω == 0
      ∧ (∃j: 1 ≤ j ≤ n:   e_j ist keine Variable
         ∧ isFun Π_j
         ∧ alle funktionsenthaltenden Variablen in
           e_j sind markiert)
    dann
      sei
        e' = f e_1...e_{j-1} (C^{e_j}_{Π_j} v_1...v_k) e_{j+1}...e_n
        mit v_1,...,v_k freie Variablen in e_j
        ds' = (data T_{Π_j} = C^{e_j}_{Π_j} Φ_1...Φ_k)
          △ ds
        mit Φ_i = { Ψ_i,  falls v_i mit Ψ_i markiert ist   ,   1 ≤ i ≤ k
                  { β_i,  sonst

          β_i neue Typvariablen
        fs' = (apply_{Π_j} c v_1...v_{countArgs Π_j}
                 = case c of
                   C^{e_j}_{Π_j} -> foldl (◇) e_j [v_1,...,v_{countArgs Π_j}])
            ∇ fs
      in (e', ds', fs')
    sonst (e, ds, fs)
  wobei
    f e_1...e_n :: Ω
    e_i :: Π_i, 1 ≤ i ≤ n
```

Abbildung 3.6: Transformationsregel `encode` mit

$$\texttt{add} :: \texttt{Int -> Int -> Int}$$

und `x` ist Variable erfüllt alle Bedingungen und wird übersetzt zu

$$\texttt{map}_{\Psi}\ \texttt{(}\texttt{C}^{\texttt{(add x)}}_{\texttt{Int -> Int}}\ \texttt{x) xs}.$$

Da `x` nicht markiert ist, entsteht der Datentyp

$$\texttt{data T a = }\texttt{C}^{\texttt{(add x)}}_{\texttt{Int -> Int}}\ \texttt{a}$$

und die `apply`–Funktion

$$\begin{array}{l}\texttt{apply}_{\texttt{Int -> Int}}\ \texttt{c y}\\ \quad \texttt{= case c of}\\ \qquad \texttt{C}^{\texttt{(add x)}}_{\texttt{Int -> Int}}\ \texttt{x -> add x y}.\end{array}$$

3.3 Applikation von Variablen

Nachdem alle Argumentfunktionen kodiert wurden, ist zu berücksichtigen, daß Variablen, die ursprünglich Funktionen als Werte besaßen, nun Konstruktoren besitzen. Ehemalige Applikationen von Funktionsvariablen müssen nun anders behandelt werden. Aus diesem Grunde werden die `apply`–Funktionen eingeführt.

Ziel der Transformationsregel `applVar` ist es, markierte Aufrufe von Funktionsvariablen durch Hinzufügen einer `apply`–Funktion zu ändern. Welche `apply`–Funktion aufgerufen werden muß, hängt von der Markierung der jeweiligen Funktionsvariablen ab.

Beispiel:

Der Clone

$$\texttt{map}_{\Psi}\ \texttt{f xs = [<f>}_{\Pi}\ \texttt{x | x <- xs]}$$

wird transformiert zu

$$\texttt{map}_{\Psi}\ \texttt{f xs = [apply}_{\Pi}\ \texttt{f x | x <- xs]}.$$

Voraussetzungen für Dekodierung

Im folgenden werden die Bedingungen vorgestellt, damit ein Ausdruck transformiert werden kann. Sei dazu der Term

$$\texttt{f e}_1 \ldots \texttt{e}_n$$

mit

$$\texttt{f e}_1 \ldots \texttt{e}_n :: \Omega$$

gegeben. Folgende Bedingungen müssen erfüllt sein:

```
applVar :: (Expr, [Data], [Fun]) -> (Expr, [Data], [Fun])
applVar (e, ds, fs)
  = wenn e Applikation der Form:  <f>Π e1...en
      ∧ f ist Variable ∧ countArgs Ω == 0
    dann
      sei e' = applyΠ f e1...en
      in (e', ds, fs)
    sonst (e, ds, fs)
  wobei
    f e1...en :: Ω
```

Abbildung 3.7: Definition der Transformationsregel `applVar`

(i) `countArgs` Ω `== 0`

(ii) `f` ist eine Funktionsvariable

(iii) `f` ist markiert

Die Bedingung (ii) stellt sicher, daß die `apply`–Funktion nur Variablen als Parameter hat. Funktionssymbole wurden bereits durch die Regel `encode` transformiert.

Die Forderung, daß `f` markiert sein muß (Bedingung (iii)), gewährleistet, daß die für diesen Typen richtige `apply`–Funktion verwendet wird.

Nach Bedingung (i) werden nur Funktionsapplikationen behandelt, deren Ergebnisse keine Funktionen sind. Ist dies nicht der Fall, wird die Funktionsapplikation als Parameter verwendet und somit durch die Regel `encode` kodiert.

Definition der Transformationsregel `applVar`

Sind alle Bedingungen erfüllt, so wird der Aufruf

`<f>`$_\Pi$ `e`$_1 \ldots$`e`$_n$

transformiert zu

`apply`$_\Pi$ `f e`$_1 \ldots$`e`$_n$.

Die Definition der Transformationsregel `applVar` ist in Abbildung 3.7 angegeben.

Die bisher eingeführten Regeln beschreiben den Kern der Transformation. Allerdings werden einige Ausdrücke noch nicht vollständig behandelt, wie z.B. in Listenkomprehensionen. Aufgrund lokaler Variablen, die durch die Generatoren eingeführt werden, kann es passieren, daß zwar die rechte Seite eines Generators markiert wurde, nicht aber die dazugehörige lokale Variable auf der linken Seite. Die in den Abschnitten 3.4 und 3.5 vorgestellten Transformationsregeln berücksichtigen diesen und ähnliche Fälle.

3.4 Aktualisieren der case–Zweige

Die Regel `updateBranches` führt Anpassungen in den `case`–Zweigen von Funktionen durch.

Mustervergleich (Pattern Matching) ist ein wichtiges Konstrukt in funktionaler Programmierung, bei dem ein Ausdruck mit einem Muster verglichen wird. Falls dieser Vergleich erfolgreich ist, wird der Ausdruck durch die Variablen in dem Muster aufgeteilt. Stimmt z.B. die Liste `[id, inc]` mit dem Muster `(f:fs)` überein, so enthält die Variable `f` die Funktion `id`, die Variable `fs` den Wert `[inc]`. Zu beachten ist jedoch, daß durch das Auftreten des Listenkonstruktors `(:)` der Typ von `[id, inc]` :: `[Int -> Int]` aufgeteilt wird in die Typen `f` :: `Int -> Int` und `fs` :: `[Int -> Int]`.

Beim Mustervergleich werden lokale Variablen eingeführt, die, obwohl sie funktionsenthaltend sind, nicht markiert werden, da die Haupttransformationsregeln nur die Parameter einer Funktion behandeln.

Beispiel:

Sei die Funktionsdefinition

```
mapf :: [a -> b] -> a -> [b]
mapf fs x
  = case fs of
    []     -> []
    (g:gs) -> g x : mapf gs x
```

mit dem Aufruf

```
mapf [id, inc] 0
```

gegeben.
Durch die Transformationsregel `funSpec` wird der Clone

$$
\begin{array}{l}
\texttt{mapf}_{\Psi}\ \texttt{fs x} \\
\quad \texttt{= case <fs>}_{\texttt{[Int -> Int]}}\ \texttt{of} \\
\qquad \texttt{[]} \quad\ \texttt{-> []} \\
\qquad \texttt{(g:gs) -> g x : mapf}_{\Psi}\ \texttt{gs x.}
\end{array}
$$

mit

$$\Psi\ \texttt{= [Int -> Int] -> Int -> [Int]}$$

erzeugt. Das funktionsenthaltende Argument `fs` wurde zwar markiert, jedoch befindet sich in dem zweiten `case`–Zweig die Funktionsvariable `g`, die ebenfalls noch markiert werden muß, damit die Regel `applVar` folgenden Code erzeugen kann:

$$
\begin{array}{l}
\texttt{mapf}_{\Psi}\ \texttt{fs x} \\
\quad \texttt{= case fs of} \\
\qquad \texttt{[]} \quad\ \texttt{-> []} \\
\qquad \texttt{(g:gs) -> apply}_{\texttt{Int -> Int}}\ \texttt{g x : mapf}_{\Psi}\ \texttt{gs x.}
\end{array}
$$

Dies wird durch die Transformationsregel `updateBranches` sichergestellt.

Voraussetzungen für die Aktualisierung von `case`–Ausdrücken

Sei der `case`-Ausdruck

$$
\begin{array}{l}
\texttt{case e of} \\
\quad \texttt{p}_1 \ \texttt{->}\ \texttt{e}_1 \\
\qquad \vdots \\
\quad \texttt{p}_n \ \texttt{->}\ \texttt{e}_n
\end{array}
$$

mit

$$\texttt{e} :: \Pi$$

gegeben. Folgende Bedingungen müssen erfüllt sein:

(i) `containsFun` Π und

(ii) alle funktionsenthaltenden Variablen in `e` sind markiert

Definition der Transformationsregel `updateBranches`

Sind beide Voraussetzungen erfüllt, so werden alle funktionsenthaltenden Variablen, die im Muster auftreten, markiert. Der `case`–Ausdruck

$$
\begin{array}{l}
\texttt{case e of} \\
\quad \texttt{C}_1 \ \texttt{x}_{(1,1)} \ldots \texttt{x}_{(1,m_1)} \ \texttt{->}\ \texttt{e}_1 \\
\qquad \vdots \\
\quad \texttt{C}_n \ \texttt{x}_{(n,1)} \ldots \texttt{x}_{(n,m_n)} \ \texttt{->}\ \texttt{e}_n
\end{array}
$$

mit

$$\texttt{x}_{(\xi,\zeta)} :: \Pi_{(\xi,\zeta)}, \ \ 1 \leq \xi \leq n, \ \ 1 \leq \zeta \leq m_\xi$$

wird transformiert zu

$$
\begin{array}{l}
\texttt{case (unmark e) of} \\
\quad \texttt{C}_1 \ \texttt{x}_{(1,1)} \ldots \texttt{x}_{(1,m_1)} \ \texttt{->}\ \texttt{e}_1\texttt{'} \\
\qquad \vdots \\
\quad \texttt{C}_n \ \texttt{x}_{(n,1)} \ldots \texttt{x}_{(n,m_n)} \ \texttt{->}\ \texttt{e}_n\texttt{'}
\end{array}
$$

mit

$$\texttt{e}_i\texttt{'} \ \texttt{=} \ \texttt{e}_i \ \ \texttt{[<x}_{(i,j_1)}\texttt{>}_{\Pi_{(i,j_1)}}\texttt{/x}_{(i,j_1)}\texttt{,}\ldots\texttt{,<x}_{(i,j_{p_i})}\texttt{>}_{\Pi_{(i,j_{p_i})}}\texttt{/x}_{(i,j_{p_i})}\texttt{]}$$

und

$$1 \leq i \leq n.$$

```
updateBranches :: (Expr, [Data], [Fun]) -> (Expr, [Data], [Fun])
updateBranches (e, ds, fs)
  = wenn e ist ein case-Ausdruck der Form:
      case expr of
        C_1 x_(1,1) ... x_(1,m_1) -> e_1
          ⋮
        C_n x_(n,1) ... x_(n,m_n) -> e_n
      ∧ containsFun Π
      ∧ alle funktionsenthaltenden Variablen in expr
        sind markiert
    dann
      sei
        e' =
          case expr of
            C_1 x_(1,1) ... x_(1,m_1) -> e_1'
              ⋮
            C_n x_(n,1) ... x_(n,m_n) -> e_n'
      in (e', ds, fs)
    sonst (e, ds, fs)
  wobei
    expr :: Π
    x_(ξ,ζ) :: Π_(ξ,ζ)  (1 ≤ ξ ≤ n, 1 ≤ ζ ≤ m_ξ)
    e_i' = e_i  [<x_(i,j_1)>Π_(i,j_1)/x_(i,j_1), ..., <x_(i,j_{p_i})>Π_(i,j_{p_i})/x_(i,j_{p_i})]
          (j_l Indizes der funktionsenthaltenden Variablen
           im Muster des i-ten case-Zweigs, 1 ≤ l ≤ p_i)
```

Abbildung 3.8: Definition der Transformationsregel `updateBranches`

Die Indizes j_l, $(1 \leq l \leq p_i)$ sind die der funktionsenthaltenden Variablen im Muster des i-ten `case`–Zweigs.

Im Gegensatz zu der in [BBH97] benutzten Sprache besitzt in $\mathcal{HDC}$ jeder Ausdruck einen Typ. Insofern erübrigt sich eine Typherleitung für die $\Pi_{(i,j_l)}$, man kann direkt alle funktionsenthaltenden Variablen in den Ausdrücken markieren (Abbildung 3.8).

Beispiel:

Der Clone

```
mapf_Ψ fs x
  = case <fs>_[Int -> Int] of
      []      -> []
      (g:gs) -> g x : mapf_Ψ gs x
```

wird durch `updateBranches` transformiert zu

```
mapf_Ψ fs x
  = case fs of
      []      -> []
      (g:gs) -> <g>_Int -> Int x : mapf_Ψ <gs>_[Int -> Int] x.
```

3.5 Aktualisieren der Listenkomprehensionen

Ähnliche Probleme wie in `case`–Ausdrücken treten auch in Listenkomprehensionen auf, da hier ebenfalls lokale Variablen eingeführt werden, die funktionsenthaltend sein können.

Beispiel:

Sei folgender Clone gegeben:

```
mapf_Ψ fs x = [f x | f <- <fs>_[Int -> Int]].
```

Die Funktionsvariable `f` muß markiert werden, damit die Regel `applVar` erfolgreich angewendet werden kann, aber nur `fs` wurde durch die Regel `funSpec` markiert.

Voraussetzungen für die Aktualisierung von Listenkomprehensionen

Sei

$$\texttt{[e | gf}_1\texttt{,}\ldots\texttt{,gf}_n\texttt{]}$$

eine Listenkomprehension. Folgende Voraussetzung muß erfüllt sein:

$$\begin{aligned}&(\exists j{:}\, 1 \leq j \leq n{:}\ \ (\texttt{gf}_j \text{ ist ein Generator } \texttt{x <- xs} \text{ mit } \texttt{xs} :: \texttt{[}\Pi\texttt{]})\\ &\ \wedge (\texttt{containsFun}\ \Pi)\\ &\ \wedge (\text{alle funktionsenthaltenden Variablen in } \texttt{xs} \text{ sind markiert}))\end{aligned}$$

```
updateLC :: (Expr, [Data], [Fun]) -> (Expr, [Data], [Fun])
updateLC (e, ds, fs)
  = wenn e Listenkomprehension der Form:  [expr | gf_1,...,gf_n]
      ∧ (∃j : 1 ≤ j ≤ n : gf_j ist Generator x <- xs, xs :: [Π]
         ∧ containsFun Π
         ∧ alle funktionsenthaltenden Variablen
           in xs sind markiert)
    dann
      sei
        expr' = expr [<x_j>_Π_j/x_j]
        gf_l' = gf_l[<x_j>_Π_j/x_j], j < l ≤ n
        e' = [expr' | gf_1,...,gf_{j-1},
                      x <- (unmark xs),
                      gf_{j+1}',...,gf_n']
      in (e', ds, fs)
    sonst (e, ds, fs)
```

Abbildung 3.9: Definition der Transformationsregel `updateLC`

Definition der Transformationsregel `updateLC`

Ist die Bedingung erfüllt, so wird die Listenkomprehension

```
[e | gf_1,...,gf_{j-1}, x <- xs, gf_{j+1},...,gf_n]
```

transformiert zu

```
[e' | gf_1,...,gf_{j-1}, x <- (unmark xs), gf_{j+1}',...,gf_n']
```

mit

```
e' = e [<x>_Π/x]
```

und

```
gf_l' = gf_l [<x>_Π/x]
```

für $j < l \leq n$.

Zu beachten ist, daß die Variable `x` nur in Ausdrücken $\texttt{gf}_l$ für $j < l \leq n$ auftreten kann. Die Definition der Funktion `updateLC` ist in Abbildung 3.9 dargestellt. Diese Regel existiert in [BBH97] nicht, da die Inputsprache ohne Listenkomprehensionen definiert wurde.

3.6 Benutzerdefinierte Datentypen

Treten in benutzerdefinierten Datentypen Funktionstypen auf, so kann hier nicht geklont werden, sondern man muß diese Datentypen generalisieren. Die grundlegende Idee der Generalisierung ist, daß Funktionstypen durch eine neue Typvariable ersetzt werden.

Beispiel:

Der Datentyp

```
data T a b = C (a -> b)
```

kann generalisiert werden zu

```
data T c = C c.
```

Die Typvariablen `a` und `b` werden nicht mehr benötigt, sie können entfernt werden.

Der in [BBH97, S. 33] aufgeführte Algorithmus ist jedoch unvollständig, weil der Fall von nicht-regulären Datentypen nicht berücksichtigt wurde. Ein *nicht-regulärer Datentyp* [BM98] ist ein parametrisierter rekursiver Datentyp, der im rekursiven Aufruf unterschiedliche Instanzen von Typparametern besitzt.

Beispiel:

Der Datentyp

```
data T a b = E | C (a -> b) (T b a)
```

ist nicht regulär, da gegenüber der linken Seite die Reihenfolge der Typvariablen vertauscht ist.

Im folgenden werden die Schwierigkeiten vorgestellt, die die Generalisierung von nicht-regulären benutzerdefinierten Datentypen mit sich bringt. Sei dazu der algebraische Datentyp

```
data T a b = E | C (a -> b) (T b a)
```

gegeben. Da der Konstruktor `C` den Funktionstyp `(a -> b)` hat, muß dieser durch eine neue Typvariable `c` ersetzt werden.

Ansatz I

Die neu entstandene Typvariable `c` wird zu den restlichen Typvariablen hinzugefügt. Zu beachten ist, daß jedes Vorkommen von `T` um die Typvariable `c` erweitert werden muß.

Beispiel:

Aus obigem Datentyp wird nun

```
data T a b c = E | C c (T b a c).
```

Problem: Das Vertauschen der Variablen `a` und `b` geht verloren, da kein Zusammenhang zwischen `a`, `b` und `c` besteht. Der Ausdruck

```
C int2bool (C bool2int E),
```

der bei der ursprünglichen Datentypdefinition den Typ

```
T Int Bool
```

besitzt, verursacht nun einen Typfehler, da versucht wird, die Typvariable `c` zum einen mit `Int -> Bool` und zum anderen mit `Bool -> Int` zu belegen.

Ansatz II

Ersetze rekursive Vorkommen des Datentypen durch eine neue Typvariable.

Beispiel:

Obiger Datentyp wird generalisiert zu

```
data T c d = E | C c d.
```

Der Ausdruck

```
C int2bool (C bool2int E)
```

hat nun den Typ

```
T (Int -> Bool) (T (Bool -> Int) (T a b)),
```

jedoch treten hier Schwierigkeiten bei der Typbestimmung auf.
Sei dazu folgende Funktionsdefinition gegeben:

```
myId fs
  = case fs of
    E      -> E
    C g gs -> C g (myId gs).
```

Anhand der Konstruktoren würde sich für `myId` der Typ `T c d -> T c e` ergeben, da `g` :: `c` und `gs` :: `d`. Der rekursive Aufruf von `myId` müßte deswegen den Typ `d -> e` besitzen. Diese beiden Typausdrücke lassen sich jedoch nicht miteinander unifizieren; ein unendlicher Typ würde entstehen, da `c` in `T c d` und `e` in `T c e` auftritt.

Ansatz III

Wie man erkennen kann, bereitet das Vertauschen von Typvariablen Probleme. Wäre die Reihenfolge der Variablen im rekursiven Aufruf und im Kopf gleich, dann könnte Ansatz I erfolgreich angewendet werden.

Dazu erweitert man vor der Generalisierung den Datentypen um zusätzliche Konstruktoren, die die Vertauschung von Typvariablen simulieren, ohne daß im rekursiven Aufruf die Reihenfolge der Typvariablen verändert wird.

Beispiel:

Aus dem algebraischen Datentypen

```
data T a b = E | C (a -> b) (T b a)
```

wird folgender neuer algebraischer Datentyp erzeugt:

$$
\begin{aligned}
&\texttt{data T a b = E} \\
&\qquad\qquad\;\texttt{| C (a -> b) (T a b)} \\
&\qquad\qquad\;\texttt{| } \texttt{C}_{\texttt{a/b}}\texttt{ (b -> a) (T a b).}
\end{aligned}
$$

Die Reihenfolge der Typvariablen im Kopf und im rekursiven Aufruf ist gleich, die Vertauschung der Typen wird durch den Konstruktor $\texttt{C}_{\texttt{a/b}}$ durchgeführt. Nun kann die Generalisierung nach Ansatz I durchgeführt werden.
Problem: Das Programm muß um zusätzliche Mustervergleiche erweitert werden; wichtig ist ebenfalls eine Umbenennung bestimmter Konstruktoren.

Beispiel:

Der Aufruf

```
C int2bool (C bool2int E)
```

muß nun übersetzt werden zu

$$\texttt{C}_{\texttt{a/b}}\ \texttt{int2bool (C bool2int E)}$$

oder zu

$$\texttt{C int2bool (}\texttt{C}_{\texttt{a/b}}\ \texttt{bool2int E).}$$

Der Funktion `myId` muß ein zusätzlicher `case`–Zweig angefügt werden

```
myId fs
  = case fs of
    C g gs -> C g (myId gs)
    C_a/b g gs -> C_a/b g (myId gs).
```

Weitere Probleme treten auf, wenn im rekursiven Aufruf nicht nur die Typvariablen vertauscht sind, sondern Typvariablen mehrfach auftreten oder durch andere Typen ersetzt werden:

```
data T a b = E | C (a -> b) (T a a)
```

bzw.

```
data T a b = E | C (a -> b) (T a Int).
```

Implementierte Lösung

Um die mit der Generalisierung verbundenen Probleme zu vermeiden, sind benutzerdefinierte Datentypen, die Funktionstypen besitzen, nicht erlaubt, falls sie entweder nicht regulär oder verschränkt rekursiv sind.

Ist der Datentyp regulär, so kann Ansatz I angewendet werden. Die zweite Einschränkung verhindert Probleme beim Aktualisieren der Typvariablen.

Zusätzlich werden in Konstruktordefinitionen auftretende benutzerdefinierte Datentypen durch eine neue Typvariable ersetzt. Dies erleichtert den Vorgang der Generalisierung, da sich sonst Änderungen in den Typvariablen kaskadenartig fortsetzen. Die Transformationsregel für die Generalisierung ist in Abbildung 3.10 angegeben.

Da sich durch die Generalisierung Typinformationen ändern, muß sie nach Anwendung der übrigen Transformationsregeln durchgeführt werden.

```
generalizeArrows :: [Data] -> [Data]
generalizeArrows [] = []
generalizeArrows (d : ds)
  = wenn d enthält Konstruktor mit Funktionstyp
    dann
      wenn d nicht regulär ∨ d verschränkt rekursiv
      dann error ‘‘cannot generalize!’’
      sonst
        sei
          d0 = ersetze alle Funktionstypen
            und andere benutzerdefinierte Datentypen in d
            durch eine neue Typvariable
          aktualisiere Typvariablen:
          d1 = μ d0
        in d1 : generalizeArrows ds
    sonst d : generalizeArrows ds
```

Abbildung 3.10: Definition der Transformationsregel `generalizeArrows`

3.7 Entfernen von übriggebliebenen Marken

Die Regel `updateCon` in [BBH97, S. 33] sieht vor, nach der Entfunktionalisierung verbleibende Markierungen aus Konstruktoren zu entfernen. Marken, die nicht in Konstruktoren auftraten, wurden durch die Typspezialisierungsregel entfernt, was sich jedoch als fehlerhaft erwiesen hat (vgl. Abschnitt 3.1). Die Markierungen werden deshalb nicht entfernt und sind nun nach erfolgreicher Entfunktionalisierung ebenfalls vorhanden. Eine Transformationsregel `removeMark` entfernt daher alle Marken.

Wird die Markierung, wie in $\mathcal{HDC}$, nicht textuell durchgeführt, sondern in der internen Repräsentation von Ausdrücken (`Expr`), so kann `removeMark` weggelassen werden.

3.8 Vorverarbeitung: η-Erweiterung

Um sicherzustellen, daß die Elimination von Funktionen höherer Ordnung auch wirklich alle Funktionen transformiert, muß vorher noch eine η-Erweiterung (Abbildung 3.11) durchgeführt werden, die Funktionsdefinitionen um soviel Argumente erweitert, daß die Anzahl der Parameter in der Typdefinition und in der Funktionsdefinition gleich sind. Bei [BBH97] wurde dieser Fall nicht berücksichtigt.

Beispiel:

Sei folgende Funktion gegeben

```
etaExpand :: [Fun] -> [Fun]
etaExpand [] = []
etaExpand ((f, \x1 ->...-> \xn -> D) : fs)
  = sei
      diff = countArgs Π - n
    in
      wenn diff > 0
      dann
        (f, \x1 ->...-> \xn -> \v1 ->...-> \vdiff
          -> (foldl (◇) D [v1,...,vdiff])) : etaExpand fs
      sonst
        f : etaExpand fs
  wobei
    f :: Π
```

Abbildung 3.11: η-Erweiterung

```
foo :: [Int] -> [Int]
foo = map inc.
```

Diese Funktion kann von `funSpec` nicht spezialisiert werden, da

`countArgs (map inc)` $\neq$ `0`.

Führt man jedoch vor der Elimination von Funktionen höherer Ordnung eine η-Erweiterung auf der Funktionsdefinition durch, so erhält man

```
foo :: [Int] -> [Int]
foo eta = map inc eta.
```

Jetzt sind alle Bedingungen erfüllt, damit die Regel `funSpec` angewendet werden kann.

3.9 Bemerkungen

In [BBH97] übersah man bei der Elimination von Funktionen höherer Ordnung die Probleme im Zusammenhang mit Typvariablen. So kann man während der Transformation den Typ eines Konstruktors nicht korrekt angeben.

Beispiel:

Seien die Funktion

```
second a b = b
```

und die Ausdrücke

```
map id [1..10]
```

und

```
map (second x) [1..10]
```

gegeben. Sowohl `id` als auch `(second x)` besitzen beide im Kontext den Typ `Int -> Int`. Kodiert man zuerst `id` durch $\mathtt{C}^{\mathtt{id}}_{\mathtt{Int\ ->\ Int}}$, so hat der Konstruktor den Typ $\mathtt{T}_{\mathtt{Int\ ->\ Int}}$.
Fährt man fort mit der Kodierung von `second x` durch ($\mathtt{C}^{\mathtt{(second\ x)}}_{\mathtt{Int\ ->\ Int}}$ `x`), so hat dieser Ausdruck wegen `x` :: `a` den Typ $\mathtt{T}_{\mathtt{Int\ ->\ Int}}$ `a` und der vorher bestimmte Typ von $\mathtt{C}^{\mathtt{id}}_{\mathtt{Int\ ->\ Int}}$ ist nun falsch.

Da sich algebraische Datentypen im Verlauf der Transformation ändern, werden neue Typvariablen für die Typen der Konstruktoren eingeführt, eine Typüberprüfung muß nach der Elimination von Funktionen höherer Ordnung durchgeführt werden.

Der in diesem Kapitel angegebene Algorithmus terminiert: Die Transformationsregel `funSpec` erzeugt Kopien von Funktionen mit spezielleren Typen. Da polymorphe Funktionen im Programm nur mit einer endlichen Menge von Typen instanziiert werden, erzeugt `funSpec` eine endliche Menge von Kopien. Die Regel `encode` kodiert Argumentfunktionen durch Argumente erster Ordnung. Sind alle Argumentfunktionen ersetzt, so läßt sich `encode` nicht mehr anwenden. Durch `applVar` wird aus einer Applikation einer Funktionsvariablen eine Applikation einer `apply`–Funktion, die die Funktionsvariable als Parameter besitzt. Applikationen von Funktionsvariablen werden eliminiert und deswegen terminiert auch die Regel `applVar`.

Nach erfolgreicher Anwendung der Transformationsregeln entsteht ein entfunktionalisiertes Programm, dessen Typen erneut berechnet werden müssen. Die Eingabesprache mußte jedoch bei der Verwendung von algebraischen Datentypen eingeschränkt werden. Aus diesem Grund wurde die Implementierung dieser Entfunktionalisierung in $\mathcal{HDC}$ ersetzt durch die im nächsten Kapitel vorgestellte Modifikation.

Kapitel 4

Entfunktionalisierung in $\mathcal{HDC}$

In diesem Kapitel werden Modifikationen angegeben, die notwendig sind, um die Elimination von Funktionen höherer Ordnung in den $\mathcal{HDC}$–Compiler einzubinden.

Zur Erzeugung von parallelem C–Code und zur Größeninferenz ist es wichtig, daß der abstrakte Zwischencode nicht nur entfunktionalisiert, sondern auch monomorph ist; polymorphe Funktionen sind in C nicht zulässig. Dies erfordert neben der Entfunktionalisierung eine zusätzliche Monomorphisierung.

Findet diese noch vor der Entfunktionalisierung statt, so sind die Probleme mit den Typvariablen aus den Abschnitten 3.6 und 3.9 beseitigt, d.h. die Transformation ist nun für eine größere Menge benutzerdefinierter Datentypen durchführbar. Die Typüberprüfung nach der Entfunktionalisierung kann entfallen, da sich die Typen der hinzugefügten Konstruktoren während der Transformation nicht ändern; Typinformationen von modifizierten Ausdrücken lassen sich direkt bestimmen.

Eine ausführliche Beschreibung der Monomorphisierung befindet sich in Abschnitt 4.1. Die Änderungen, die an der Entfunktionalisierung vorzunehmen sind, werden in Abschnitt 4.2 vorgestellt.

Daraus ergibt sich das Schema aus Abbildung 4.1 für die modifizierte Entfunktionalisierung.

4.1 Monomorphisierung

Zur Monomorphisierung eines Programms beginnt man mit einem Ausdruck, der bereits monomorph ist, also keine Typvariablen mehr enthält, und spezialisiert alle darin enthaltenen Ausdrücke rekursiv.

Die Ausführung von Programmen in $\mathcal{HDC}$ erfordert eine Funktion `parmain`, die den Typ `[Int] -> [Int]` besitzt. Ausgehend von dieser Funktion wird das Programm monomorphisiert.

Beispiel:

Dem Programm

```
parmain :: [Int] -> [Int]
```

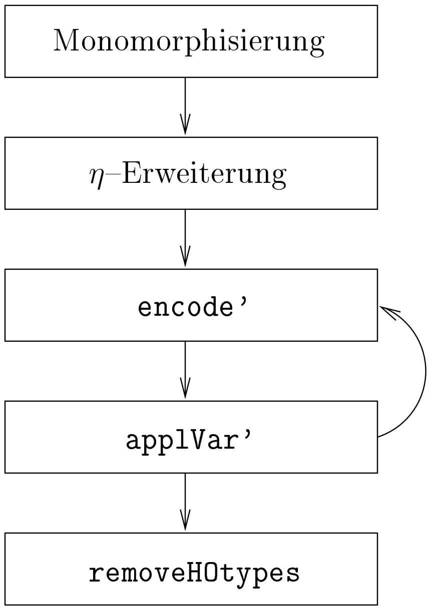

Abbildung 4.1: Die Phasen der modifizierten Higher-order Elimination

```
parmain xs = map id xs

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

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

werden ähnlich wie in der Transformationsregel `funSpec` Kopien mit spezielleren Typen hinzugefügt:

$$\begin{array}{l}
\texttt{map}_{\Psi} :: \texttt{(Int -> Int) -> [Int] -> [Int]} \\
\texttt{map}_{\Psi}\ \texttt{f xs = [f x | x <- xs]} \\
\\
\texttt{id}_{\texttt{Int -> Int}} :: \texttt{Int -> Int} \\
\texttt{id}_{\texttt{Int -> Int}}\ \texttt{x = x}
\end{array}$$

mit

$$\Psi\ \texttt{= (Int -> Int) -> [Int] -> [Int]}.$$

Die Funktion `parmain` wird transformiert zu

$$\begin{array}{l}
\texttt{parmain} :: \texttt{[Int] -> [Int]} \\
\texttt{parmain xs = map}_{\Psi}\ \texttt{id}_{\texttt{Int -> Int}}\ \texttt{xs}.
\end{array}$$

Der Typ der Clones ergibt sich aus der jeweiligen Anwendungsstelle. Die Funktion wird kopiert und die Typen werden mittels Unifikation und Substitution spezialisiert.

Der auf diese Weise spezialisierte Code ist aber nicht unbedingt monomorph; es können noch Typvariablen existieren, für die keine Bindung existiert. Diese werden durch den Datentyp `Void` ersetzt.

Beispiel:

Sei folgendes Programm gegeben

```
parmain :: [Int] -> [Int]
parmain xs = map (second id) xs

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

second :: a -> b -> b
second a b = b

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

In der Funktion `parmain` besitzt `map` den speziellen Typen

```
((a -> a) -> Int -> Int) -> [Int] -> [Int],
```

der die Typvariable `a` enthält.
Das Programm wird daher transformiert zu

```
data Void = Void

parmain :: [Int] -> [Int]
parmain xs = map_Ψ (second_Φ id_Void -> Void) xs

map_Ψ :: ((Void -> Void) -> Int -> Int) -> [Int] -> [Int]
map_Ψ f xs = [f x | x <- xs]

second_Φ :: (Void -> Void) -> Int -> Int
second_Φ a b = b

id_Void -> Void :: Void -> Void
id_Void -> Void x = x
```

mit

```
Ψ = ((Void -> Void) -> Int -> Int) -> [Int] -> [Int]
```

und

```
Φ = (Void -> Void) -> Int -> Int.
```

Die Monomorphisierung beschränkt sich dabei nicht nur auf Funktionen, sondern bearbeitet auch benutzerdefinierte Datentypen.

Anhand der Anwendungsstelle eines Konstruktors lassen sich die Typvariablen in dem algebraischen Datentyp, zu dem der Konstruktor gehört, instanziieren. Man kann nun eine Kopie dieses Datentyps anlegen, verbleibende Typvariablen auf der rechten Seite der Definition des Datentyps wieder durch `Void` ersetzen, und anschließend alle Parameter des Datentyps entfernen.

Beispiel:

Aus dem benutzerdefinierten Datentypen

```
data T a b = E | C (a -> b) (T b a)
```

mit dem Ausdruck

```
C int2bool (C bool2int E)
```

entstehen die zwei neuen Datentypen

$$\texttt{data } \texttt{T}_{\texttt{Int, Bool}} = \texttt{E}_{\texttt{Int, Bool}} \mid \texttt{C}_{\texttt{Int, Bool}} \texttt{ (Int -> Bool) } \texttt{T}_{\texttt{Bool, Int}}$$

und

$$\texttt{data } \texttt{T}_{\texttt{Bool, Int}} = \texttt{E}_{\texttt{Bool, Int}} \mid \texttt{C}_{\texttt{Bool, Int}} \texttt{ (Bool -> Int) } \texttt{T}_{\texttt{Int, Bool}}$$

mit dem modifizierten Ausdruck

$$\texttt{C}_{\texttt{Int, Bool}} \texttt{ int2bool (}\texttt{C}_{\texttt{Bool, Int}} \texttt{ bool2int } \texttt{E}_{\texttt{Int, Bool}}\texttt{)}.$$

Die Probleme aus Abschnitt 3.6 sind dadurch beseitigt. Allerdings existiert eine Klasse von nicht-regulären Datentypen, die nicht monomorphisiert werden können.

Beispiel:

Seien folgende benutzerdefinierte Datentypen gegeben:

```
data Nest a = NilN | ConsN a (Nest (a, a))
data Bush a = NilB | ConsB a (Bush (Bush a)).
```

Eine Monomorphisierung dieser Datentypen würde nicht terminieren, da durch den rekursiven Aufruf immer wieder neue Datentypen entstünden im Gegensatz zum Datentyp

```
data T a b = E | C a (T b a).
```

Datentypen, die durch den rekursiven Aufruf Typparameter vergrößern, müssen deshalb ausgeschlossen werden. Diese Einschränkung reduziert die Menge der transformierbaren Programme jedoch nicht allzu stark, da fast keine sinnvollen Funktionen existieren, die auf diesen Datentypen definiert werden können (vgl. [BM98]). Das Ergebnis der Monomorphisierung mit dieser Einschränkung ist ein zum Eingabeprogramm semantisch äquivalentes monomorphes Programm.

Abschließend werden alle algebraischen Datentypen intern zu einem einzigen Datentyp `DATA` vereinigt; dies ist möglich, da sie nun keine Typvariablen mehr besitzen.

Dieses Zusammenfassen vereinfacht die spätere Implementation in C durch die Verwendung von durchnumerierten Konstruktoren (vgl. [JL92]).

Ebenso wie in Kapitel 3 muß vor der nun folgenden modifizierten Entfunktionalisierung eine η-Erweiterung durchgeführt werden (vgl. Abschnitt 3.8).

4.2 Modifizierte Entfunktionalisierung

Im Anschluß an die Monomorphisierung und η-Erweiterung wird eine modifizierte Elimination von Funktionen höherer Ordnung durchgeführt.

Bei der Analyse der Transformationsregeln stellt man fest, daß die vor der Entfunktionalisierung durchgeführte Monomorphisierung der Typspezialisierung aus Abschnitt 3.1 ähnelt. Im Gegensatz dazu werden jedoch die Funktionen ausgehend von einer monomorphen Funktion bearbeitet. Wurden in [BBH97] nur Funktionen höherer Ordnung so weit wie nötig spezialisiert, so werden nun alle Funktionen monomorphisiert. Dadurch erübrigt sich das Markieren von Ausdrücken ebenso wie die Überprüfungen, ob ein Ausdruck markiert ist. Die Transformationsregeln `funSpec`, `updateArms`, `updateLC` und `removeMark` werden deswegen nicht mehr benötigt.

In der Entfunktionalisierung aus Kapitel 3 bereitete das Vorhandensein von Typvariablen Schwierigkeiten bei der Typbestimmung transformierter Ausdrücke (vgl. Abschnitt 3.9). Da nach der Monomorphisierung keine Typvariablen mehr existieren und durch die Kodierung von Argumentfunktionen keine weiteren erzeugt werden, kann die Typbestimmung des entfunktionalisierten Programms nun während der Transformation stattfinden. Ebenso kann das Generalisieren von Datentypen höherer Ordnung entfallen; Funktionstypen werden durch die jeweiligen algebraischen Datentypen ersetzt, die diese Typen repräsentieren.

Nur die Transformationsregeln `encode` und `applVar` werden noch benötigt und müssen daher neu definiert werden (Abschnitte 4.2.1 und 4.2.2), sowie eine zusätzliche Regel `removeHOtypes` in Abschnitt 4.2.3.

4.2.1 Modifizierte Kodierung

Sei der Ausdruck

$$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n$$

mit

$$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n :: \Omega$$

und

$$(\forall j : 1 \leq j \leq n : \texttt{e}_j :: \Pi_j)$$

gegeben. Damit die Kodierung von Funktionen in Konstruktoren durchgeführt werden kann, müssen folgende Bedingungen erfüllt sein:

(i) $\texttt{countArgs}\ \Omega$ `== 0`

(ii) $(\exists j\colon 1 \leq j \leq n\colon (\texttt{isFun}\ \Pi_j) \wedge (\texttt{e}_j$ ist keine Variable$))$

(iii) `f` ist entweder ein Clone, eine `apply`-Funktion oder ein Konstruktor

Sind alle Bedingungen erfüllt, so wird der Ausdruck

$$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_{j-1}\ \ \texttt{e}_j\ \ \texttt{e}_{j+1} \ldots \texttt{e}_n$$

mit

$$\texttt{isFun}\ \Pi_j$$

und

$$\texttt{f} :: \Pi$$

transformiert zu

$$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_{j-1}\ \ (\texttt{C}_{\Pi_j}^{\texttt{e}_j} \texttt{v}_1 \ldots \texttt{v}_k)\ \ \texttt{e}_{j+1} \ldots \texttt{e}_n$$

mit

$$\texttt{C}_{\Pi_j}^{\texttt{e}_j} \texttt{v}_1 \ldots \texttt{v}_k :: \texttt{DATA}$$

und $\texttt{v}_1, \ldots, \texttt{v}_k$ freie Variablen in $\texttt{e}_j$.
Der algebraische Datentyp `DATA` wird erweitert um diesen Konstruktor zu

$$\texttt{data DATA = } \ldots \texttt{|}\ \ \texttt{C}_{\Pi_j}^{\texttt{e}_j}\ \ \Phi_1 \ldots \Phi_k$$

mit $\texttt{v}_l :: \Phi_l\ (1 \leq l \leq k)$ und die folgende `apply`-Funktion wird erzeugt bzw. um den neuen `case`–Zweig erweitert

$$\begin{array}{l}
\texttt{apply}_{\Pi_j} :: \texttt{DATA -> } \Pi_j \\
\texttt{apply}_{\Pi_j}\ \ \texttt{c}\ \ \texttt{x}_1 \ldots \texttt{x}_{\texttt{countArgs}\ \Pi_j} \\
\quad \texttt{= case c of} \\
\qquad \texttt{C}_{\Pi_j}^{\texttt{e}_j} \texttt{v}_1 \ldots \texttt{v}_k\ \ \texttt{-> foldl}\ (\diamond)\ \texttt{e}_j\ \ [\texttt{x}_1, \ldots, \texttt{x}_{\texttt{countArgs}\ \Pi_j}].
\end{array}$$

Der modifizierte Algorithmus zur Kodierung aus Abbildung 4.2 unterscheidet sich von dem aus Abschnitt 3.2 durch das Fehlen der Überprüfung der Markierung und durch das modifizierte Hinzufügen der Konstruktoren. Zusätzlich werden für alle neu eingeführten Ausdrücke deren Typen abgeleitet.

4.2.2 Modifizierte Variablenapplikation

Im folgenden werden die Bedingungen vorgestellt, die gelten müssen, damit ein Ausdruck um eine Dekodierfunktion erweitert werden kann. Sei dazu der Ausdruck

$$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n$$

mit

```
encode' :: (Expr, [Data], [Fun]) -> (Expr, [Data], [Fun])
encode' (e, ds, fs)
  = wenn e Applikation der Form:  f e_1...e_n
      ∧ f ist entweder Clone, apply-Funktion oder Konstruktor
      ∧ countArgs Ω == 0
      ∧ (∃j: 1 ≤ j ≤ n: e_j ist keine Variable
         ∧ isFun Π_j)
    dann
      sei
        e' = f e_1...e_{j-1} (C^{e_j}_{Π_j} v_1...v_k) e_{j+1}...e_n
        mit v_1,...,v_k freie Variablen in e_j
        berechne Typ des kodierten Ausdrucks:
        C^{e_j}_{Π_j} v_1...v_k :: DATA
        ds' = (data DATA = C^{e_j}_{Π_j} Φ_1...Φ_k)
          △ ds
        fs' = (apply_{Π_j} :: DATA -> Π_j
               apply_{Π_j} c v_1...v_{countArgs Π_j}
                 = case c of
                   C^{e_j}_{Π_j} -> foldl (◇) e_j [v_1,...,v_{countArgs Π_j}])
            ∇ fs
      in (e', ds', fs')
    sonst (e, ds, fs)
  wobei
    f e_1...e_n :: Ω
    e_i :: Π_i,  1 ≤ i ≤ n
    v_l :: Φ_l,  1 ≤ l ≤ k
```

Abbildung 4.2: modifizierte Kodierung

```
applVar' :: (Expr, [Data], [Fun]) -> (Expr, [Data], [Fun])
applVar' (e, fs, ds)
  = wenn e Applikation der Form:  f e₁...eₙ
      ∧ f ist Variable ∧ countArgs Ω == 0
    dann
      sei e' = apply_Π f e₁...eₙ
      in (e', ds, fs)
    sonst (e, ds, fs)
  wobei
    f e₁...eₙ :: Ω
    f :: Π
```

Abbildung 4.3: modifizierte Dekodierung

$$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n :: \Omega$$

gegeben. Folgende Punkte müssen erfüllt sein:

(i) `countArgs` Ω `== 0`

(ii) `f` ist eine Funktionsvariable

Der Ausdruck

$$\texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n$$

mit `f` $:: \Pi$ wird durch die modifizierte Regel aus Abbildung 4.3 transformiert zu

$$\texttt{apply}_\Pi\ \texttt{f}\ \texttt{e}_1 \ldots \texttt{e}_n.$$

Diese beiden Transformationsregeln werden solange auf das Eingabeprogramm ausgeführt, bis sich keine Änderungen mehr ergeben. Das Resultat ist ein entfunktionalisiertes Programm, das allerdings noch Typinkonsistenzen aufweist.

4.2.3 Entfernen von Funktionstypen

Da sich durch obige Regeln lokal Typen ändern, sind die Typinformationen nun nicht mehr konsistent. Unter anderem werden Argumentfunktionen durch Konstruktoren kodiert und besitzen daher einen anderen Typ als vor der Transformation; Funktionstypen müssen in dem Ausdruck, in dem kodiert wurde, aktualisiert werden.

Beispiel:

Gegeben sei folgendes Programm

```
parmain :: [Int] -> [Int]
parmain xs = map id xs

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

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

Nach der Monomorphisierung und der Anwendung der modifizierten Regeln entsteht daraus das Programm

$$
\begin{array}{l}
\texttt{data DATA = } \texttt{C}^{\texttt{id}}_{\texttt{Int -> Int}} \\
\\
\texttt{parmain :: [Int] -> [Int]} \\
\texttt{parmain xs = } \texttt{map}_{\Psi}\ \texttt{C}^{\texttt{id}}_{\texttt{Int -> Int}}\ \texttt{xs} \\
\\
\texttt{map}_{\Psi}\ \texttt{:: (Int -> Int) -> [Int] -> [Int]} \\
\texttt{map}_{\Psi}\ \texttt{f xs = [}\texttt{apply}_{\texttt{Int -> Int}}\ \texttt{f x | x <- xs]} \\
\\
\texttt{id}_{\texttt{Int -> Int}}\ \texttt{:: Int -> Int} \\
\texttt{id}_{\texttt{Int -> Int}}\ \texttt{x = x}
\end{array}
$$

mit

$$\Psi \texttt{ = (Int -> Int) -> [Int] -> [Int].}$$

Die Definition von $\texttt{map}_{\Psi}$ ist jedoch wegen

```
f :: DATA
```

falsch getypt; sie muß geändert werden zu

$$\texttt{map}_{\Psi} \texttt{ :: DATA -> [Int] -> [Int].}$$

Diese Typänderungen könnten auch während der modifizierten Entfunktionalisierung durchgeführt werden. Der Aufwand wäre aber groß, da Ausdrücke in einer Baumstruktur abgelegt sind. Ändern sich in einem Ausdruck am Blatt des Baumes (`Expr`) die Typen, so müßten die Typinformation im kompletten Baum aktualisiert werden.

Aus Effizienzgründen wird daher der Algorithmus `removeHOtypes` in Abbildung 4.4 nach der Entfunktionalisierung auf alle Ausdrücke angewendet.

```
removeHOtypes :: Type -> Type
removeHOtypes t
  = falls t von der Form
    (a -> b)  ⟹ DATA
    T t₁...tₙ  ⟹ T (removeHOtypes t₁)...(removeHOtypes tₙ)
                 mit T Datentyp
    andernfalls  ⟹ t
```

Abbildung 4.4: Entfernen von Typen höherer Ordnung

Kapitel 5

Bewertung

5.1 Vergleich der beiden Transformationen

Im folgenden werden die Transformationen aus Kapitel 3 und 4 miteinander verglichen. Kriterien sind Codegröße, Transformationsdauer, Anwendbarkeit der Transformationen und Unterschiede in der Ausführungszeit der entfunktionalisierten Programme.

Im $\mathcal{HDC}$–Compiler können phasenweise Transformationen durchgeführt und die Zwischenergebnisse mit dem Interpreter ausgeführt werden. Zur Bewertung der transformierten Programme wurde der Interpreter um statistische Ausgaben erweitert. Dazu zählen u.a. die Berechnung der Anzahl der Reduktionen und die Anzahl der Funktionsaufrufe, die nötig sind, um ein Programm auszuwerten.

Wird im Interpreter eine Funktion aufgerufen, so erhöht sich die Zahl der Funktionsaufrufe um 1. Der Interpreter reduziert zur Auswertung eines Ausdrucks diesen solange, bis keine Reduktion mehr möglich ist. Daraus ergibt sich die Anzahl der Reduktionen. Zur Bestimmung der Codegröße wird die Anzahl der Funktionen und der Definitionen algebraischer Datentypen bestimmt, die zur Auswertung von `parmain` benötigt werden.

Wir wählten zwei Benchmarks: `quicksort`, eine Implementation des Quicksort-Algorithmus und `karatsuba`, ein Programm, das zwei durch Koeffizientenlisten gegebene Polynome miteinander multipliziert. Zum Beispiel wird das Polynom $x + 1$ durch die Koeffizientenliste `[1,1]` und das Polynom x durch `[1,0]` dargestellt. Die Polynomkoeffizienten werden der Funktion `parmain` in einer Liste übergeben, die in der Mitte durch die Funktionen `left` und `right` in die für `karatsuba` benötigten zwei Listen geteilt wird. Die Eingabe `[1,1,1,0]` wird in die Listen `[1,1]` und `[1,0]` geteilt und diese anschließend der Funktion `karatsuba` übergeben, was der der Multiplikation von $x + 1$ mit x entspricht. Die beiden Programme befinden sich im Anhang B.

Die Ergebnisse werden in den Abbildungen 5.1, 5.2, 5.3 und 5.4 präsentiert. Dabei steht „HOE“ für die Entfunktionalisierung aus Kapitel 3 und „mod. HOE“ für deren Modifikation. Mit „Original“ wird das Inputprogramm für die Programmtransformation bezeichnet, d.h. ein Programm ohne syntaktischen Zucker.

	Original	HOE	mod. HOE
Anzahl Funktionen	21	29	32
Anzahl Datentypen	0	10	1

Abbildung 5.1: Vergleich: Codegröße `karatsuba`

	Original	HOE	mod. HOE
Anzahl Funktionen	11	15	15
Anzahl Datentypen	0	4	1

Abbildung 5.2: Vergleich: Codegröße `quicksort`

`Eingabe [1,1,1,0]`	Original	HOE	mod. HOE
Anzahl Reduktionen	1397	1803	1803
Anzahl Funktionsaufrufe	300	406	406

Abbildung 5.3: Vergleich: Effizienz `karatsuba`

`Eingabe [9,0,4,1,7,5,3,2,8,6]`	Original	HOE	mod. HOE
Anzahl Reduktionen	2286	2994	2994
Anzahl Funktionsaufrufe	473	675	675
`Eingabe [9..0]`	Original	HOE	mod. HOE
Anzahl Reduktionen	3136	4040	4040
Anzahl Funktionsaufrufe	645	903	903
`Eingabe [19..0]`	Original	HOE	mod. HOE
Anzahl Reduktionen	10461	12345	12345
Anzahl Funktionsaufrufe	2105	2643	2643

Abbildung 5.4: Vergleich: Effizienz `quicksort`

5.1.1 Codegröße

Ziel in [BBH97] ist es, neben der Entfunktionalisierung die Größe des transformierten Programms so minimal wie möglich zu halten. Dazu werden im Algorithmus nur die Typvariablen der funktionsenthaltenden Argumente spezialisiert; alle anderen Typvariablen bleiben erhalten.

Im Gegensatz dazu findet vor der modifizierten Entfunktionalisierung eine Monomorphisierung des Programms statt. Diese bewirkt, daß von jeder Funktion für jeden Typ, der benötigt wird, eine Kopie angelegt wird. Beinhaltet ein Programm viele polymorphe Funktionen, die mit unterschiedlichen Typen verwendet werden, so wächst das Programm.

Das Zusammenfassen aller Konstruktoren zu dem Datentyp `DATA` hilft jedoch, das Programm zu verkürzen. Es existiert nur noch eine Datentypdefinition anstelle von mehreren in [BBH97].

5.1.2 Dauer der Transformationen

Beim Vergleich der beiden Verfahren bezüglich ihrer Transformationsdauer stellt man fest, daß die modifizierte Version um eine Monomorphisierung erweitert ist, d.h. zusätzliche Schritte zur Entfunktionalisierung nötig sind. Durch die Monomorphisierung sind jedoch die Transformationsregeln für die Funktionsspezialisierung überflüssig geworden; dadurch benötigt die modifizierte Entfunktionalisierung weniger Iterationen als die ursprüngliche.

Zusätzlich kann bei der modifizierten Entfunktionalisierung die Typüberprüfung nach abgeschlossener Transformation entfallen. Die in Kapitel 3 vorgestellte Lösung ist daher bezüglich der Transformationsdauer ineffizienter als die modifizierte Version.

5.1.3 Anwendbarkeit der Transformationen

Wie bereits in Abschnitt 3.6 erwähnt, treten in [BBH97] Probleme bei algebraischen Datentypen mit Funktionstypen auf. Die Sprache für transformierbare Programme muß daher eingeschränkt werden. Die modifizierte Transformation läßt hingegen eine größere Menge von Eingabeprogrammen zu.

5.1.4 Ausführungszeit der Programme

Trotz unterschiedlicher Größe der transformierten Programme sind die Anzahl der Reduktionen und Funktionsaufrufe gleich. Dies ist auch nicht weiter verwunderlich, da in beiden Fällen Argumentfunktionen kodiert und durch `apply`–Funktionen ausgeführt werden. Wird nach der Transformation aus [BBH97] eine Funktion polymorph benutzt, muß sie trotzdem so oft wie benötigt aufgerufen werden. Bei der modifizierten Version werden dazu die je nach Typ unterschiedlichen Kopien der Funktionen aufgerufen. Da die Anzahl der Reduktionen der Ausführungszeit eines Programms entspricht, ist die Ausführungszeit eines durch [BBH97] entstandenen Programms identisch mit der eines durch die modifizierte Entfunktionalisierung entstandenen.

5.1.5 Folgerung

Anhand der obigen Ergebnisse stellt man fest, daß die modifizierte Entfunktionalisierung der ursprünglichen vorzuziehen ist. Sie stellt weniger Einschränkungen an das Eingabeprogramm und erzeugt den entfunktionalisierten Code effizienter. Da für die Codegenerierung in $\mathcal{HDC}$ ohnehin monomorpher Code benötigt wird, kommt der Vorteil der Polymorphie des Verfahrens in [BBH97] nicht zur Geltung.

5.2 Laufzeit entfunktionalisierter Programme

In diesem Abschnitt wird untersucht, inwiefern sich die Laufzeit der Programme durch eine Entfunktionalisierung ändert. Wie schon in Abschnitt 5.1 festgestellt wurde, unterscheiden sich die beiden Methoden zur Entfunktionalisierung nicht in der Laufzeit der transformierten Programme; die nachfolgenden Ergebnisse sind also für beide Methoden gültig. Als Benchmarks wurden die Programme `quicksort`, `karatsuba`, `sumcps`, `sample1` und `sample2` aus Anhang B verwendet. Das aus Abschnitt 1.3.1 übernommene Programm `sumcps` berechnet die Summe einer gegebenen Liste im sogenannten *Continuation-Passing Style* [App92]; `sample1` und `sample2` sind aus [BBH97] entnommen. Die Laufzeiten werden für eine sequentielle (Abschnitt 5.2.1) und parallele Ausführung (5.2.2) berechnet; Ergebnisse der Messung befinden sich in Tabelle 5.5. Dabei bezeichnen $t^{\mathtt{seq}}_{\mathtt{ORIG}}$ und $t^{\mathtt{seq}}_{\mathtt{HOE}}$ die sequentielle Ausführungszeit für ein Programm vor und nach der Entfunktionalisierung, $t^{\mathtt{par}}_{\mathtt{ORIG}}$ und $t^{\mathtt{par}}_{\mathtt{HOE}}$ die parallele Ausführungszeit.

5.2.1 Sequentielle Ausführung des transformierten Programms

Wird ein entfunktionalisiertes Programm sequentiell ausgeführt, so entsteht keine Verbesserung in der Laufzeit gegenüber dem ursprünglichen Programm; das Programm ist zwar nun von erster Ordnung, jedoch verhält es sich wie das urspüngliche: Wurde vor der Transformation eine Funktion als Argument übergeben und anschließend ausgeführt, so wird nun ein Repräsentant der Funktion verwendet, der zuerst dekodiert werden muß, bevor die dazugehörige Funktion ausgeführt werden kann.

Diese Dekodierung von Argumentfunktionen durch `apply`–Funktionen verursacht zusätzliche Reduktionen und Funktionsapplikationen.

5.2.2 Parallele Ausführung des transformierten Programms

Zur Messung der Ausführungszeit im parallelen Fall wurde der Interpreter im $\mathcal{HDC}$–Compiler um die Berechnung eines freien Schedules erweitert. Im *freien Schedule* werden Auswertungen von Ausdrücke so früh wie möglich (as soon as possible, ASAP) geplant. Zusätzlich werden bestimmte Ausdrücke mit Zeitkosten versehen, die deren Bearbeitungsaufwand darstellen:

Ist ein Ausdruck eine Applikation $(\mathtt{e}_l\ \mathtt{e}_r)$, so werden $\mathtt{e}_l$ und $\mathtt{e}_r$ gleichzeitig ausgewertet. Die Auswertungen von $\mathtt{e}_l$ und $\mathtt{e}_r$ sind zu den Zeitpunkten t_l und t_r beendet und somit ist das Ergebnis dieser Applikation zum Zeitpunkt

Programm `karatsuba`				
Eingabe	$t^{\mathtt{seq}}_{\mathtt{ORIG}}$	$t^{\mathtt{par}}_{\mathtt{ORIG}}$	$t^{\mathtt{seq}}_{\mathtt{HOE}}$	$t^{\mathtt{par}}_{\mathtt{HOE}}$
`[1,1,1,0]`	1397	88	1803	95
Programm `quicksort`				
Eingabe	$t^{\mathtt{seq}}_{\mathtt{ORIG}}$	$t^{\mathtt{par}}_{\mathtt{ORIG}}$	$t^{\mathtt{seq}}_{\mathtt{HOE}}$	$t^{\mathtt{par}}_{\mathtt{HOE}}$
`[9..0]`	3136	166	4040	195
Programm `sample1`				
Eingabe	$t^{\mathtt{seq}}_{\mathtt{ORIG}}$	$t^{\mathtt{par}}_{\mathtt{ORIG}}$	$t^{\mathtt{seq}}_{\mathtt{HOE}}$	$t^{\mathtt{par}}_{\mathtt{HOE}}$
`[1..3]`	146	29	301	41
`[1..5]`	466	95	1303	143
`[1..8]`	3202	664	10702	1048
Programm `sample2`				
Eingabe	$t^{\mathtt{seq}}_{\mathtt{ORIG}}$	$t^{\mathtt{par}}_{\mathtt{ORIG}}$	$t^{\mathtt{seq}}_{\mathtt{HOE}}$	$t^{\mathtt{par}}_{\mathtt{HOE}}$
`[1..3]`	140	19	173	20
`[1..5]`	222	25	277	26
`[1..10]`	427	40	537	41
Programm `sumcps`				
Eingabe	$t^{\mathtt{seq}}_{\mathtt{ORIG}}$	$t^{\mathtt{par}}_{\mathtt{ORIG}}$	$t^{\mathtt{seq}}_{\mathtt{HOE}}$	$t^{\mathtt{par}}_{\mathtt{HOE}}$
`[1..3]`	88	21	139	25
`[1..5]`	138	31	217	37
`[1..10]`	263	56	412	67

Abbildung 5.5: Ausführungszeiten ausgewählter Programme

Ausdruck	Kosten
`case`	$\lceil \log_2(\text{Anzahl der Zweige}) \rceil$
`if-then-else`	1
Applikation	1
primitiver Operator	1
Suche nach Funktionsdefinitionen	1
sonst	0

Abbildung 5.6: Kosten für Berechnung des freien Schedules

$\max(t_l, t_r) + 1$ vorhanden, weil die Kosten einer Applikation als 1 angenommen werden.

Die Kosten für die Suche nach einer Funktionsdefinition, ebenso wie die Kosten für einen primitiven Operator werden mit 1 angesetzt. Zu den primitiven Operatoren zählen Funktionen wie `(+)`, `(-)`, `(*)`, `(/)`, etc. Für Listenoperationen gelten zur Zeit ebenfalls Kosten 1.

Wird ein `case`–Ausdruck ausgewertet, so entstehen Kosten von $\lceil \log_2 n \rceil$, wobei n die Anzahl der `case`–Zweige ist. Ein `if-then-else`–Ausdruck wird sequentiell ausgewertet, d.h. zuerst die Bedingung überprüft und dann je nach Ergebnis der `then`- oder `else`-Teil mit zusätzlichen Kosten 1 ausgeführt.

Alle übrigen Ausdrücke sind mit Kosten 0 versehen. In Tabelle 5.6 werden die Kosten für die jeweiligen Ausdrücke aufgelistet.

5.2.3 Folgerung

Da durch die Entfunktionalisierung die Laufzeitcharakteristik eines transformierten Programms der eines Programms höherer Ordnung ähnlich ist und zusätzlich Funktionen zur Dekodierung von Argumentfunktionen benötigt werden, verschlechtert sich die Laufzeit eines entfunktionalisierten Programms gegenüber dem ursprünglichen. Dies kommt im sequentiellen Fall mehr zu Geltung als im parallelen Fall. Der durch die Entfunktionalisierung entstandene Code läßt sich jedoch noch optimieren, um effizientere Programme zu erhalten.

Kapitel 6

Ausblick

In den nächsten Abschnitten werden Modifikationen der Entfunktionalisierung vorgestellt, die die Laufzeit transformierter Programme verbessern sollen. Dazu zählen zum einen die Erweiterung der Transformation um eine Behandlung von `let`- und λ-Ausdrücken (Abschnitt 6.1), zum anderen eine zusätzliche Vorverarbeitungsphase, die Argumentfunktionen direkt in die Funktionsdefinition von Funktionen höherer Ordnung kodiert (Abschnitt 6.2).

6.1 Entfunktionalisierung für `let`- und λ-Ausdrücke

Die nun vorgestellten Modifikationen der Entfunktionalisierung sind sowohl auf die Version aus Kapitel 3 als auch auf die aus Kapitel 4 anwendbar; das Ziel, das erreicht werden soll, ist eine Laufzeitverbesserung transformierter Programme.

Entfunktionalisierung von λ-Ausdrücken

Da λ-Ausdrücke namenlose Funktionen sind, können sie nicht geklont bzw. weiter spezialisiert werden. Lediglich die Regeln `encode` und `applVar` müssen λ-Ausdrücke behandeln. Bei der Entfunktionalisierung aus Kapitel 3 muß zusätzlich dafür gesorgt werden, daß funktionsenthaltende Parameter in λ-Ausdrücken markiert werden.

Die Regel `applVar` muß für die Behandlung von λ-Ausdrücken angepaßt werden, damit Applikationen von Funktionsvariablen richtig behandelt werden.

Beispiel:

Der Ausdruck

```
(\f xs -> [f x | x <- xs]) inc [1..10]
```

wird durch die Regel `encode` transformiert zu

$$\texttt{(\textbackslash f xs -> [f x | x <- xs])}\ \texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}\ \texttt{[1..10]}$$

und muß durch `applVar` geändert werden in

$$\texttt{(\textbackslash f xs -> [apply}_{\texttt{Int -> Int}}\texttt{ f x | x <- xs])}$$
$$\texttt{C}^{\texttt{inc}}_{\texttt{Int -> Int}}\ \texttt{[1..10].}$$

Wird ein λ-Ausdruck einer Funktion als Parameter übergeben, so kann dieser wie gewohnt durch einen Konstruktorausdruck ersetzt werden; die freien Variablen des λ-Ausdrucks müssen dabei berücksichtigt werden (Transformationsregel `encode`).

Beispiel:

In der Funktion

```
sum f ys
  = case ys of
    []     -> f 0
    (x:xs) -> sum (\n -> x + f n) xs
```

wird die Argumentfunktion `(\n -> x + f n)` mit den freien Variablen `x` und `f` durch den Konstruktorausdruck

$$\texttt{(C}^{\texttt{(\textbackslash n -> x + f n)}}_{\texttt{Int -> Int}}\ \texttt{x f)}$$

ersetzt.
Die für diesen Konstruktor erzeugte `apply`–Funktion wird folgendermaßen definiert:

$$\begin{array}{l}\texttt{apply}_{\texttt{Int -> Int}}\ \texttt{c y}\\ \quad\texttt{= case c of}\\ \qquad\texttt{C}^{\texttt{(\textbackslash n -> x + f n)}}_{\texttt{Int -> Int}}\ \texttt{x f -> (\textbackslash n -> x + apply}_{\texttt{Int -> Int}}\ \texttt{f n) y}\end{array}$$

und kann modifiziert werden zu

$$\begin{array}{l}\texttt{apply}_{\texttt{Int -> Int}}\ \texttt{c y}\\ \quad\texttt{= case c of}\\ \qquad\texttt{C}^{\texttt{(\textbackslash n -> x + f n)}}_{\texttt{Int -> Int}}\ \texttt{x f -> x + apply}_{\texttt{Int -> Int}}\ \texttt{f y.}\end{array}$$

Entfunktionalisierung in `let`–Ausdrücken

Ebenso lassen sich Programme mit `let`–Ausdrücken entfunktionalisieren. Dabei werden wie gewohnt alle Transformationsregeln, sowie die η-Erweiterung und bei der Transformation aus Kapitel 4 die Monomorphisierung, durchgeführt mit Berücksichtigung von lokalen Funktionsdefinitionen.

Lokale Funktionen bereiten jedoch Schwierigkeiten, wenn sie in `apply`–Funktionen außerhalb ihres Gültigkeitsbereiches verwendet werden.

Beispiel:

Sei

```
test :: [a] -> [a]
test
  = let h = id
    in map h
```

mit dem Aufruf

```
test [1..10]
```

gegeben. Durch Monomorphisierung und η-Erweiterung entsteht daraus

$$\begin{array}{l}
\mathtt{test}_{\mathtt{[Int]\ ->\ [Int]}} :: \mathtt{[Int]\ ->\ [Int]} \\
\mathtt{test}_{\mathtt{[Int]\ ->\ [Int]}}\ \mathtt{eta}_1 \\
\quad \mathtt{=\ let\ h}_{\mathtt{Int\ ->\ Int}}\ \mathtt{eta}_2\ \mathtt{=\ id}_{\mathtt{Int\ ->\ Int}}\ \mathtt{eta}_2 \\
\qquad \mathtt{in\ map}_{\Psi}\ \mathtt{h}_{\mathtt{Int\ ->\ Int}}\ \mathtt{eta}_1 \\
\\
\mathtt{map}_{\Psi} :: \mathtt{(Int\ ->\ Int)\ ->\ [Int]\ ->\ [Int]} \\
\mathtt{map}_{\Psi}\ \mathtt{f\ xs\ =\ [f\ x\ |\ x\ <-\ xs]}
\end{array}$$

mit Ψ `= (Int -> Int) -> [Int] -> [Int]`.
Die Entfunktionalisierung erzeugt daraus

$$\begin{array}{l}
\mathtt{test}_{\mathtt{[Int]\ ->\ [Int]}} :: \mathtt{[Int]\ ->\ [Int]} \\
\mathtt{test}_{\mathtt{[Int]\ ->\ [Int]}}\ \mathtt{eta}_1 \\
\quad \mathtt{=\ let\ h}_{\mathtt{Int\ ->\ Int}}\ \mathtt{eta}_2\ \mathtt{=\ id}_{\mathtt{Int\ ->\ Int}}\ \mathtt{eta}_2 \\
\qquad \mathtt{in\ map}_{\Psi}\ \mathtt{C}^{\mathtt{h}}_{\mathtt{Int\ ->\ Int}}\ \mathtt{eta}_1 \\
\\
\mathtt{map}_{\Psi} :: \mathtt{DATA\ ->\ [Int]\ ->\ [Int]} \\
\mathtt{map}_{\Psi}\ \mathtt{f\ xs\ =\ [apply}_{\mathtt{Int\ ->\ Int}}\ \mathtt{f\ x\ |\ x\ <-\ xs].}
\end{array}$$

Bei der Definition der `apply`–Funktion treten nun Probleme auf, da die Funktion `h` lokal in `test` definiert wurde; `h` muß nun entweder global (durch λ-Lifting) oder lokal in der `apply`–Funktion definiert werden.

Der Vorteil der Benutzung gemeinsamer Teilausdrücke geht jedoch dabei verloren und damit ebenso eine Steigerung der Effizienz.

6.2 Verwandte Arbeit

In [CD96] wird ein Verfahren zur Entfunktionalisierung vorgestellt, das formale Parameter einer Funktion höherer Ordnung entfernt, wenn deren aktuelle Parameter Argumentfunktionen sind. Eine Argumentfunktion wird direkt in die Funktionsdefinition einer Funktion höherer Ordnung geschrieben anstelle der Variablen.

Beispiel:

Aus dem Aufruf

```
map inc xs
```

wird

$\mathtt{map}_{\mathtt{inc}}\ \mathtt{xs}$

mit der Definition

$\mathtt{map}_{\mathtt{inc}}\ \mathtt{xs\ =\ [inc\ x\ |\ x\ <-\ xs].}$

In diesem Fall verbessert sich die Laufzeit des Programms, da keine Konstruktoren durch `apply`–Funktionen dekodiert werden müssen. Die Programmgröße nimmt dadurch allerdings zu.

Im Gegensatz zu der in dieser Arbeit vorgestellten Entfunktionalisierung muß die in [CD96] vorgestellte stark eingeschränkt werden: Argumentfunktionen dürfen nur eliminiert werden, wenn die variable-only–Bedingung erfüllt ist.

Definition: variable-only

Seien f_i, $1 \leq i \leq n$, verschränkt rekursive Funktionen mit

$$\mathtt{f}_i\ \ \mathtt{x}_1 \ldots \mathtt{x}_{m_i}\ \ \mathtt{=}\ \ \mathtt{D}_i.$$

Der j-te Parameter $\mathtt{x}_j$ $(1 \leq j \leq m_i)$ der i-ten Funktion $\mathtt{f}_i$ heißt *variable-only*, genau dann wenn jeder in $\mathtt{D}_k$, $(1 \leq k \leq n)$, auftretende rekursive Aufruf von $\mathtt{f}_i$

$$\mathtt{f}_i\ \ \mathtt{e}_1 \ldots \mathtt{e}_{m_i},$$

an j-ter Position nur aus einem formalen Parameter besteht, also

$$\mathtt{e}_j \in \{\mathtt{x}_1, \ldots, \mathtt{x}_{m_i}\}.$$

Zu beachten ist, daß bei jeder nicht-rekursiven Funktion jeder Parameter trivialerweise variable-only ist.

Beispiel:

Alle Parameter in der Definition von `map` sind variable-only. Aus diesem Grund kann die Argumentfunktion direkt in die Definition kodiert werden.
Im Gegensatz dazu ist bei der Funktion `sum` aus Abschnitt 1.3.1 der erste Parameter nicht variable-only, er wird im rekursiven Aufruf verändert. Deshalb kann diese Funktion nicht transformiert werden.

Definition der Transformation

Sei

$$\mathtt{f}\ \ \mathtt{e}_1 \ldots \mathtt{e}_{j-1}\ \ \mathtt{e}_j\ \ \mathtt{e}_{j+1} \ldots \mathtt{e}_n$$

eine Funktionsapplikation mit

$$\mathtt{e}_j :: \Pi$$

und es gilt:

(i) `f` ist Funktionssymbol mit $\mathtt{f}\ \ \mathtt{x}_1 \ldots \mathtt{x}_n\ \ \mathtt{=}\ \ \mathtt{D}$

(ii) $\mathtt{e}_j$ ist Funktionssymbol
$\vee$ in $\mathtt{e}_j$ treten keine funktionsenthaltenden Variablen auf

(iii) $\mathtt{isFun}\ \Pi$

(iv) $\mathtt{x}_j$ erfüllt variable-only Bedingung

Transformation	Anzahl Reduktionen
keine	119
HOE	200
variable-only	116

Abbildung 6.1: Vergleich der Methoden aus [BBH97] und [CD96]

Dann wird

$\mathtt{f}\ \mathtt{e}_1 \ldots \mathtt{e}_{j-1}\ \mathtt{e}_j\ \mathtt{e}_{j+1} \ldots \mathtt{e}_n$

transformiert zu

$\mathtt{f}_{\mathtt{e}_j}\ \mathtt{e}_1 \ldots \mathtt{e}_{j-1}\ \mathtt{v}_1 \ldots \mathtt{v}_k\ \mathtt{e}_{j+1} \ldots \mathtt{e}_n$

mit $\mathtt{v}_1, \ldots, \mathtt{v}_k$ freie Variablen in $\mathtt{e}_j$, und die Definition

$\mathtt{f}_{\mathtt{e}_j}\ \mathtt{x}_1 \ldots \mathtt{x}_{j-1}\ \mathtt{v}_1 \ldots \mathtt{v}_k\ \mathtt{x}_{j+1} \ldots \mathtt{x}_j$
$\quad = \mathtt{D}\ [\mathtt{e}_j/\mathtt{x}_j]$

wird hinzugefügt.

Beispiel:

Sei der Aufruf

```
map (add x) [1..10]
```

gegeben. Die Argumentfunktion `(add x)` darf entfernt werden, da alle Bedingungen erfüllt sind. Daraus ergibt sich der modifizierte Aufruf

$\mathtt{map}_{\mathtt{(add\ x)}}\ \mathtt{x\ [1..10]}$

mit

$\mathtt{map}_{\mathtt{(add\ x)}}\ \mathtt{x\ ys = [add\ x\ y\ |\ y \texttt{<-} ys]}.$

Die Einschränkung durch Bedingung (ii) stellt sicher, daß nur definierte Funktionen entfernt werden und keine Funktionsvariablen. Ist $\mathtt{e}_j$ ein komplizierterer Ausdruck, so muß überprüft werden, ob darin freie funktionsenthaltende Variablen auftreten. Ist dies der Fall, so ist das Entfernen von $\mathtt{e}_j$ nicht von Vorteil, da durch die Transformation erneut funktionsenthaltende Parameter entstünden.

Wendet man diese Transformation zusätzlich vor der in dieser Arbeit vorgestellten Entfunktionalisierung an, so erhält man ein Programm erster Ordnung, das nun ein besseres Laufzeitverhalten besitzt. Tabelle 6.1 zeigt dies anhand des Ausdrucks `map inc [1..10]`, der in $\mathcal{HDC}$ ohne Entfunktionalisierung, nach der Entfunktionalisierung (HOE) und nach der in diesem Abschnitt vorgestellten Transformation (variable-only) ausgewertet wurde. Die letztere Programmtransformation wurde dabei per Hand durchgeführt.

Da Skelette eine feste Schittstellte besitzen, darf diese Vorverarbeitung nicht auf Skelettfunktionen durchgeführt werden.

Kapitel 7

Schlußfolgerungen

Funktionen höherer Ordnung ermöglichen es, wiederverwendbare und ausdrucksstarke Programme zu schreiben. Als Skelette bieten sie zudem eine Schnittstelle zu einer effizienten vordefinierten Implementierung. Sie sind deshalb wichtiger Bestandteil der funktionalen Programmierung. Durch ihre Mächtigkeit bereiten sie jedoch Probleme bei der Analyse zur Optimierung von funktionalen Programmen. Aus diesem Grund sind viele Programmtransformationen nur für Programme erster Ordnung durchführbar.

Ein weiterer entscheidender Nachteil von Funktionen höherer Ordnung besteht in der Übersetzung funktionaler Sprachen in imperative Sprachen, die dieses Konzept kaum bzw. nicht unterstützen.

In dem Projekt $\mathcal{HDC}$ am Lehrstuhl für Programmierung wird ein Compiler für die Generierung paralleler C–Programme aus einer Haskell-ähnlichen Sprache entwickelt. Funktionen höherer Ordnung müssen in $\mathcal{HDC}$ daher sowohl zur Codegenerierung als auch zur Optimierung eliminiert werden.

In dieser Arbeit wurde die Methode der Entfunktionalisierung aus [BBH97] vorgestellt, die auf einer Idee in [Rey72] beruht, in der Argumentfunktionen durch Werte erster Ordnung kodiert werden. Fehler in den Transformationsregeln aus [BBH97] wurden verbessert und die Verwendung von algebraischen Datentypen mußte auf reguläre Datentypen eingeschränkt werden.

Zur Anwendung der Entfunktionalisierung in $\mathcal{HDC}$ wurde die Transformation modifiziert. Die Hauptprobleme in [BBH97] stellen Typvariablen und benutzerdefinierte Datentypen dar. Findet vor der modifizierten Entfunktionalisierung eine Monomorphisierung statt, lösen sich die Probleme mit Typvariablen. Da $\mathcal{HDC}$ zur Codegenerierung monomorphe Programme benötigt, mußten jedoch algebraische Datentypen ausgeschlossen werden, bei denen die Monomorphisierung nicht terminiert. Die anschließend stattfindende modifizierte Entfunktionalisierung erwies sich besser als die in [BBH97].

Durch die vorgestellte Entfunktionalisierung entsteht jedoch keine Laufzeitverbesserung gegenüber dem ursprünglichen Programm, da ein transformiertes Programm die Laufzeitcharakteristik von Programmen höherer Ordnung nachahmt. Die Laufzeit wird stark beeinträchtigt durch die zusätzliche Ausführung von `apply`–Funktionen. Die Einführung von Clones und `apply`–Funktionen führt zu Code–Expansion. Ein weiterer Nachteil dieser Art von Entfunktionalisierung besteht darin, daß nur geschlossene Programme transformiert werden können, also keine separate Compilierung möglich ist. Man bräuchte ein Modulkonzept mit Exportdeklaration von Funktionen für bestimmte Typen.

Kombiniert man die Methode der Entfunktionalisierung in [BBH97] mit der in [CD96], so erhält man effizientere Programme erster Ordnung.

Anhang A

Implementierung

A.1 Allgemeines

Die Algorithmen für die Entfunktionalisierung sind, wie alle bisherigen Module des $\mathcal{HDC}$–Compilers auch in Haskell implementiert. Der Compiler wurde zusätzlich um folgende Module erweitert:

`HOelimination.hs:`	Implementation der Transformationsregeln
`Monomorph.hs:`	Implementation der Monomorphisierung
`Shared.hs:`	Hilfsfunktionen für die Monomorphisierung und Entfunktionalisierung
`General.hs:`	allgemeine Hilfsfunktionen

In einem globalen Zustand werden alle wichtigen Informationen über den Übersetzungszustand, wie z.B. alle top-level Funktionsdefinitionen, gespeichert. Diesem Zustand wurden für die Entfunktionalisierung zusätzliche Felder angefügt:

```
data TS =
 TS { fs::FUNDECS,            -- list of all top-level
                              --   function definitions
      primfs::FUNDECS,        -- list of pre-defined functions
      vs::Int,                -- variable supply
                              --   all numbers>=vs are free
      msg::String,            -- general information obtained
                              --   during compilation
      numErrors::Int,         -- how many error messages should
                              --   be printed
      errors::[String],       -- error messages in case of a failure,
                              --   empty list indicates success
      warnings::[String],     -- warnings
          .
          .
          .
   -- used for HO-elimination
      typeTable :: [String],  -- lookup-table for coding types
                              --   (represented as Strings)
      exprTable :: [String],  -- lookup-table for coding expressions
```

```
      codeChanged :: Bool,     -- flag that signals if a
                               --   transformation succeeded
      countEAppl :: Int,       -- count number of applications
                               --   when interpreting
      countRed :: Int,         -- count number of reductions
                               --   when interpreting
   -- end HO-elimination part
          .
          .
          .
      userDefTypes :: [UsrType],
      typeCheckIterations :: Int
  } deriving (Show,Eq)
```

Änderungen am Zustand werden mittels einer sogenannten *State-Transformer*–Monade [Wad92, Wad90, LJH95, HC94] vorgenommen:

```
newtype State c a = State (c -> (a,c))

type St a = State TS a

unState :: State c a -> (c -> (a,c))
unState (State x) = x

unitState :: a -> State c a
unitState a = State (\s0 -> (a,s0))

bindState :: State c a -> (a -> State c b) -> State c b
bindState m k = State (\s0 -> let (a,s1) = (unState m) s0
                                  (b,s2) = (unState (k a)) s1
                              in (b,s2))

fetchState :: State c c
fetchState = State (\s->(s,s))

updateState :: (c -> c) -> State c ()
updateState f = State (\s0 -> ((),f s0))

          .
          .
          .
```

Die State-Transformer Monade wird durch die `newtype`–Deklaration zusammen mit den Funktionen `unitState` und `bindState` definiert. Ein State-Transformer ergibt bei Eingabe eines Anfangszustands ein Tupel bestehend aus einem Wert und einem neuen Zustand.
Die Funktion `unitState` liefert einen gegebenen Wert zurück und gibt den Zustand unverändert weiter. Die Funktion `bindState` nimmt einen State-Transformer `m` :: `State c a` und eine Funktion `k` :: `a -> State c b`. Dem State-Transformer `m` wird ein Anfangszustand übergeben; dadurch entsteht ein Paar

aus einem Wert und aus einem Zwischenzustand. Die Funktion `k` wird auf diesem Wert angewendet und es entsteht ein neuer State-Transformer, der auf dem Zwischenzustand angewendet wird. Daraus ergibt sich das Ergebnis gepaart mit dem Endzustand.
Zusätzlich existieren u.a. die Funktion `fetchState`, die den aktuellen Zustand als Wert liefert und die Funktion `updateState`, die den aktuellen Zustand mittels einer Argumentfunktion verändert.
Die im Modul `Main` aufgerufenen Funktionen zur Elimination von Funktionen höherer Ordnung sind

`spec :: St TS`	zur Funktionsspezialisierung
`expandTS :: St TS`	zur η-Erweiterung
`hoEliminate :: TS -> IO TS`	zur Entfunktionalisierung

A.2 Implementation der Transformationsregeln

Im Modul `HOelimination` befinden sich die Implementationen der jeweiligen Regeln aus Kapitel 4. Die Funktionen dafür sind:

`encode :: St TS`	Implementation der Regel `encode`
`introduceApply :: St TS`	Implementation der Regel `applVar`
`removeHOtypes :: St TS`	Implementation der Regel `removeHOtypes`

Die ebenfalls im Modul `HOelimination` definierte Funktion

```
specCalls :: KindOfRule -> Expr -> St Expr
```

führt die Entfunktionalisierung je nach Regel (`KindOfRule`) auf allen Ausdrücken durch. Dabei werden die Transformationsregeln solange auf den Funktionsdefinitionen ausgeführt, bis sich keine Änderungen mehr ergeben (Feld `codeChanged` in `TS`).
Da die Darstellung von $\mathcal{HDC}$–Programmen durch den Standardzeichensatz beschränkt ist, können Typinformationen im Subskript nicht dargestellt werden. Aus diesem Grund wird jeder Typ durch eine eindeutige Nummer repräsentiert. Diese Zahl ergibt sich aus der Listenposition des im Feld `typeTable` von `TS` abgelegten Typen. Die Funktion $\texttt{id}_{\texttt{Int -> Int}}$ wird dargestellt durch `__id_Type37`, falls der Typ `Int -> Int` an Listenposition 37 in `typeTable` steht. Analog werden die durch `encode` eingeführten Konstruktoren dargestellt (Feld `exprTable`); so wird z.B. der Konstruktor $\texttt{C}^{\texttt{id}}_{\texttt{Int -> Int}}$ durch `__Expr12_Type37` dargestellt. Durch führende `_` wird gewährleistet, daß neue Namen eindeutig sind, da führende Unterstriche in $\mathcal{HDC}$ nicht erlaubt sind.
Zu beachten ist, daß im Zustand `TS` Konstruktoren als Funktionen dargestellt werden. Um die Zugehörigkeit von Konstruktoren und Datentypen effizienter zu bestimmen, wurde das Feld `userDefTypes` in `TS` eingeführt.

Anhang B

$\mathcal{HDC}$–Beispiele

B.1 $\mathcal{HDC}$ standard prelude

```
-- first part of the prelude: contains predefined Haskell functions
-- this part should only be used with the HDC system and ignored
-- if program is tested with the Haskell system

primitive (+), (-), (*), (^) :: Num a => a -> a -> a

primitive (/) :: (Num a, Num b) => a -> b -> Double

primitive (<), (<=), (>), (>=), (/=), (==) ::
 (EqOrd a, EqOrd b) => a -> b -> Bool

primitive (&&), (||) :: Bool -> Bool -> Bool

primitive div, mod :: Int -> Int -> Int

primitive min, max :: EqOrd a => a -> a -> a

primitive not :: Bool -> Bool

primitive length :: [a] -> Int

primitive undefined :: a

f . g = \x->f (g x)

fst (x,_) = x
snd (_,y) = y

map :: (a->b)->[a]->[b]
map f x = [f v | v<-x]

-- foldl should be parallelized "reduce"
-- if types a=b and f is associative
foldl :: (a->b->a) -> a -> [b] -> a
foldl f e [] = e
```

```
foldl f e (x:xs) = foldl f (f e x) xs

zip :: [a] -> [b] -> [(a,b)]
zip xs ys = [ (xs!!i,ys!!i) | i<-[0..min (length xs) (length ys) -1]]

zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f xs ys =
 [f (xs!!i) (ys!!i) | i<-[0..min (length xs) (length ys) -1]]

take :: Int -> [a] -> [a]
take n xs = [ xs!!i | i<-[0..n-1]]

drop :: Int -> [a] -> [a]
drop n xs = [ xs!!i | i<-[n..length xs -1]]

sum :: Num a => [a] -> a
sum = foldl (+) 0.0

id :: a -> a
id x = x

---------------------------------------------------------------------
-- second part of the prelude:
-- contains definitions not defined in Haskell

primitive decodeDouble10 :: (Int,Int) -> Double
-- decodeDouble10 a b = a*10^b

primitive encodeDouble10 :: Double -> (Int,Int)
-- encodeDouble10 (decodeDouble10 a b) "=" (a,b)

ilog2 :: Int -> Int           -- ceil of real log2
ilog2 n = if n<=1 then 0
                  else 1 + ilog2 ((n+1)`div`2)

left :: [a] -> [a]
left x = [ x !! i | i <- [0..length x `div` 2 - 1] ]

right :: [a] -> [a]
right x = [ x !! i | i<- [length x `div` 2 .. length x -1] ]

flatten :: [[a]] -> [a]
flatten xss = foldl (\x y -> x++y) [] xss

dc0 :: (a -> Bool) -> (a -> b) -> (a -> [a]) -> (a -> [b] -> b)
       -> a -> b
dc0 p b d c x = if p x
                   then b x
                   else c x (map (dc0 p b d c) (d x))

dc1 :: (a -> Bool) -> (a -> b) -> (a -> [a]) -> ([b] -> b) -> a -> b
dc1 p b d c x = if p x
                   then b x
```

```
                   else (c . map (dc1 p b d c) . d) x

dc2 :: (a -> b) -> (a -> [a]) -> ([b] -> b) -> Int -> a -> b
dc2 b d c n x = if n==0
                   then b x
                   else (c . map (dc2 b d c (n-1)) . d) x

check k xs
  = if length xs == k
    then xs
    else undefined -- error "degree check failed"

dc3id :: Int
      -> (a->b)
      -> (a->[a])
      -> (a->[b]->b)
      -> Int
      -> a
      -> b
dc3id degree basic divide combine = algorithm
   where algorithm levels problem =
          if levels==0
             then basic problem
             else let subprobs = check degree (divide problem)
                      partsols = map (algorithm (levels-1)) subprobs
                  in combine problem partsols
```

B.2 Definition von karatsuba

```
karatsuba :: Num a => [a] -> [a] -> [a]
karatsuba x y
  = let basic   z = [0, fst (z!!0) * snd (z!!0)]
        divide  z
          = let a = map fst z
                b = map snd z
                h = left z
                l = right z
            in [h,l,zip (zipWith (+) (left a) (right a))
                        (zipWith (+) (left b) (right b) )]
        combine z
          = (\[h,l,m] -> let mid = zipWith (-) m (zipWith (+) h l)
                         in (left h ++
                             zipWith (+) (right h) (left mid) ++
                             zipWith (+) (left l) (right mid) ++
                             right l)) z
    in dc2 basic divide combine (ilog2 (length x)) (zip x y)

parmain :: [Int] -> [Int]
parmain xs = karatsuba (left xs) (right xs)
```

B.3 Definition von quicksort

```
quicksort :: [a] -> [a]
quicksort x
  = let p xs = length xs < 2
        b xs = xs
        d xs = let pivot   = xs!!0
                   less    = [ v | v<-xs, v<pivot ]
                   greater = [ v | v<-xs, v>pivot ]
               in (less:greater:[])
        c xs ys = ys!!0 ++ [ v | v<-xs, v == xs!!0 ] ++ ys!!1
    in dc0 p b d c x

parmain :: [Int]->[Int]
parmain x = quicksort x
```

B.4 Programm sample1

```
mp :: ((a -> a) -> a -> a) -> (a -> a) -> [a] -> [a]
mp z f x
  = case x of
    []     -> []
    (x:xs) -> (f x) : (mp z (z f) xs)

db :: (a -> a) -> a -> a
db f x = f (f x)

inc :: Int -> Int
inc x = x + 1

parmain :: [Int] -> [Int]
parmain xs = mp db inc xs
```

B.5 Programm sample2

```
maph :: [a -> b] -> a -> [b]
maph fs y
  = case fs of
    []       -> []
    (f : fs) -> (f y) : (maph fs y)

add5 :: [Int] -> [Int -> Int]
add5 y
  = case y of
    []       -> []
    (x : xs) -> (k x) : (add5 xs)

k :: Int -> Int -> Int
k x z = z + 5 * x

parmain :: [Int] -> [Int]
parmain xs = maph (add5 xs) 1
```

B.6 Programm sumcps

```
sum_cps f xs
  = case xs of
    []     -> f 0
    (x:xs) -> sum_cps (\n -> x + f n) xs

parmain xs = sum_cps id xs
```

Literaturverzeichnis

[AJ89] Andrew W. Appel und Trevor Jim. Continuation-Passing, Closure-Passing Style. In *Proceedings of Principles of Programming Languages (POPL'89)*, ACM Press, Seiten 293–302. 1989.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[BBH97] Jeffrey M. Bell, Françoise Bellegarde, und James Hook. Type-driven Defunctionalization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, ACM Press, Seiten 25–37. 1997.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall Europe, zweite Auflage, 1998.

[BM98] Richard Bird und Lambert Meertens. Nested Datatypes. In *Mathematics of Program Construction (MPC '98)*, Springer Lecture Notes in Computer Science (LNCS'98), Seiten 52–67. Springer-Verlag, 1998.

[Bro92] Manfred Broy. *Informatik — Eine grundlegende Einführung, Teil I*. Springer Verlag Berlin Heidelberg, 1992.

[BW88] Richard Bird und Philip Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 1988.

[CD96] Wei-Ngan Chin und John Darlington. A higher order removal method. *Lisp and Symbolic Computation*, 9(4):287–322, 1996.

[DFH+93] John Darlington, Anthony Field, Peter Harrison, Paul Kelly, David Sharp, Qian Wu, und Ronald L. While. Parallel Programming Using Skeleton Functions. In Arndt Bode, Mike Reeve, und Gottfried Wolf, Herausgeber, *Parallel Architectures and Languages Europe (PARLE '93)*, Lecture Notes in Computer Science 694. Springer-Verlag, 1993.

[For93] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Supercomputing*, Seiten 878–883. IEEE Computer Society Press, 1993.

[Geo84] Michael Georgeff. Transformation and reduction strategies for typed lambda expressions. *ACM Trans. on Programming Languages and Systems*, 6(4):603–631, Oktober 1984.

[Gün98] Robert Günz. *Automatische Verifikation von Gleichheitsbeweisen in Haskell*. Lehrstuhl für Programmierung. Universität Passau, Fakultät für Mathematik und Informatik, September 1998. Diplomarbeit.

[HC94] Jonathan M. D. Hill und Keith Clarke. An introduction to category theory, category theory monads, and their relationship to functional programming. Technischer Bericht QMW-DCS-681, Department of Computer Science, Queen Mary & Westfield College, August 1994.

[HJW92] P. Hudak, S. Peyton Jones, und P. Wadler. Report on the programming language Haskell. In *ACM SIGPLAN Notices*, Band 27(5). 1992.

[HL97] Christoph A. Herrmann und Christian Lengauer. Parallelization of Divide-and-Conquer by Translation to Nested Loops. Technischer Bericht MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, März 1997. To appear in *J. Functional Programming*.

[JL92] Simon Peyton Jones und David Lester. *Implementing Functional Languages — A Tutorial*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 1992.

[Joh85] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jean-Pierre Jouannaud, Herausgeber, *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, Lecture Notes in Computer Science 201. Springer-Verlag, 1985.

[Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G–machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., und Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, 1994.

[KR78] Brian W. Kernighan und Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall International (UK) Ltd., 1978.

[LJH95] Sheng Liang, Mark P. Jones, und Paul Hudak. Monad transformers and modular interpreters. In *Proceedings of Principles of Programming Languages (POPL'95)*, Seiten 333–343. ACM Press, Januar 1995.

[Mic89] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. International Computer Science Series. Addison-Wesley Publishing Company, Inc., 1989.

[O'D94] John T. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.

[Rea89] Chris Reade. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley Publishing Company, Inc., 1989.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, ACM Press, Seiten 717–740. 1972.

[Sch97] Robert Schreiber. High Performance Fortran, Version 2. *Parallel Processing Letters*, 7(4):437–449, 1997.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, Inc., zweite Auflage, 1988.

[Tho96] Simon Thompson. *Haskell — The Craft of Functional Programming*. International Computer Science Series. Addison-Wesley Publishing Company, Inc., 1996.

[Wad90] Philip Wadler. Comprehending Monads. In *Conference on Lisp and Functional Programming*, Seiten 61–78. ACM Press, Juni 1990.

[Wad92] P. Wadler. Monads for Functional Programming. In M. Broy, Herausgeber, *Program Design Calculi*, NATO ASI Series F: Computer and Systems Sciences 118, Seiten 233–264. Springer-Verlag, 1992.

Eidesstattliche Erklärung

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Diese Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, im September 1998

Christian Schaller