UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Codeoptimierung für strikte funktionale Programme

Diplomarbeit

Jan Laitenberger

24.6.1999

Erstkorrektor:	Prof. Christian Lengauer, Ph.D.
Zweitkorrektor:	Prof. Dr. Werner Grass
Betreuer:	Dipl.-Inf. Christoph Herrmann

Danksagung

Mein besonderer Dank gilt Christoph Herrmann, der sich viel Zeit für die Projektbesprechungen genommen hat und mir einige Anregungen gegeben hat. Es war auch ein Vergnügen mit ihm, im Rahmen meiner Tätigkeit als wissenschaftliche Hilfskraft zusammenzuarbeiten.

Vielmals danke ich Prof. Christian Lengauer für die Vergabe eines äußerst interessanten Themas und seine wertvollen kritischen Anmerkungen. Eine große Hilfe war auch die Möglichkeit der Nutzung der Lehrstuhlinfrastruktur.

Für die Bereitstellung und Erlaubnis zur Veröffentlichung seines Programms, danke ich Zhenjiang Hu. Es zeigt besonders gut die Effizienz der Optimierung.

Schließlich möchte ich meinen Eltern dafür danken, daß sie mir dieses Studium ermöglicht haben. Ich danke auch ganz herzlich meiner besten Freundin Atina für die vielen gemeinsamen Unternehmungen, die einen wundervollen Ausgleich zu meiner Arbeit geschaffen haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Die Codeoptimierung beschäftigt sich mit der Leistungsverbesserung von Programmen. Die Leistung eines Programms kann durch das Zählen der ausgeführten Operationen und durch absolute Zeitmessungen auf einem Rechner bewertet werden. In der vorliegenden Arbeit werden Verfahren zur Codeoptimierung vorgestellt, die auch für den am Lehrstuhl für Programmierung entwickelten $\mathcal{HDC}$–Compiler implementiert wurden. Dazu wurde der Compiler um mehrere Phasen erweitert.

$\mathcal{HDC}$ ist eine strikte funktionale Programmiersprache, die sich an der Syntax von Haskell orientiert. Es wird in beschränktem Maße automatische Parallelisierung von Benutzerprogrammen unterstützt. Dazu wird der Skelettansatz verfolgt. Skelette sind Funktionen höherer Ordnung, für die effiziente Implementierung für die Zielarchitektur vorliegen.

Eine der zusätzlichen Phasen dient zur Elimination gegenseitiger Rekursion, um nachfolgende Analysen zu erleichtern. Dafür werden zwei Verfahren vorgestellt. Beide Ideen sind nicht neu. Jedoch mußte für ein Verfahren eine Lösung zur Erhaltung der Typkorrektheit entwickelt werden. Es zeigt sich, daß die Elimination nicht für alle Programme geeignet ist und die Laufzeit ungünstiger Programme deutlich verlängert.

Der Expansion von Funktionen wird bei funktionalen Sprachen ein hohes Optimierungspotential zugeschrieben. Ein naiver Ansatz, diese Transformation ohne Beachtung gemeinsamer Teilausdrücke durchzuführen, würde an der gewaltigen Codeexplosion scheitern.

Dieses Problem wird mit Einführung einer neuen Zwischencode–Ebene gelöst. Es wird für jede Funktion ein gerichteter azyklischer Graph (DAG) angelegt, in dem gleiche Ausdrücke nicht mehrfach auftreten. Diese Umwandlung erfolgt durch einen bekannten Standardalgorithmus. Damit werden viele Programme schon deutlich durch die Reduzierung von Mehrfachauswertungen verbessert.

Auf der neuen Zwischencode–Ebene wird in einer Optimierungsschleife das Programm weiter verbessert. Hier kann nun die Expansion von Funktionen kontrolliert unter Erhaltung der gemeinsamen Teilausdrücke ablaufen. Die Expansion macht über mehrere Funktionen verstreute Gemeinsamkeiten nutzbar und bringt Ausdrücke aus mehreren Funktionen zusammen. Damit wird eine weitere Verbesserung des Programms durch partielle Auswertung möglich.

Im Anschluß an die Optimierungsphase wird abstrakter Code erzeugt, der sich leicht in eine imperative Sprache, wie z.B. C, oder auch in Assembler für eine bestimmte Maschine übersetzen läßt.

Mit Beispielprogrammen wird der große Nutzen der Optimierung bestätigt: Bei einem optimierten Datamining–Algorithmus wird für eine Testeingabe die Anzahl der ausgeführten Operationen um über 40% reduziert.

1. Einleitung

1.1. Motivation

Funktionale Programmiersprachen bieten die Möglichkeit, Algorithmen knapp und präzise zu formulieren. Dabei kann sich der Benutzer voll und ganz auf die abstrakte Lösung seines Problems konzentrieren und muß sich keine Gedanken darüber machen, wie die Berechnung im Computer abläuft und der Speicher verwaltet wird.

Das hohe Abstraktionsniveau eines funktionalen Programms macht seine Übersetzung in ein effizientes Programm schwierig. Die Transformationen zur Auflösung von Hochsprachenkonstrukten erzeugen zusätzlichen Aufwand in Programmen. Dieser kann in einer Phase der Codeoptimierung wieder reduziert werden. Ebenso kann durch die Optimierung die Effizienz von Benutzerprogrammen durch Wiederverwendung von Werten gesteigert werden.

Eine gute Codeoptimierungsphase kann die Qualität des kompilierten Codes wesentlich verbessern. Sowohl die Ausführungszeit als auch die Codemenge können hier maßgeblich beeinflußt werden. Es lohnt sich also, dieser Phase besondere Aufmerksamkeit zu widmen.

In Programmen kommt es häufig vor, daß eine Definition mehrfach verwendet werden soll.

Beispiel:

```
a = ... (aufwendige Berechnung)
y = f a + g a
```

Hier werden die Funktionen `f` und `g` auf das Ergebnis einer aufwendigen Berechnung angewendet. Man bezeichnet `a` als gemeinsamen Teilausdruck von `f a` und `g a`. Von einem effizienten Compiler darf man erwarten, daß die Berechnung von `a` höchstens einmal durchgeführt wird.

Gleiche Ausdrücke, die mehrfach auftreten und gemeinsam genutzt werden können, sind möglicherweise ebenso im Quellprogramm kodiert. Andererseits ist es auch möglich, daß sie durch Programmtransformationen während der Übersetzung entstehen. Durch die Optimierung werden dann, so weit es möglich ist, Gemeinsamkeiten erkannt und eliminiert, um Mehrfachauswertungen zu vermeiden.

Funktionale Programme bestehen in der Regel aus vielen kleinen Funktionen. Dadurch ergeben sich kaum Nachteile, falls die Auswertung eines Programms

durch *Graphreduktion* [JL92, Kapitel 3] erfolgt. Wird das Programm jedoch Funktion für Funktion in eine imperative Zwischensprache übersetzt, so macht die große Anzahl von Funktionsaufrufen die Ausführung ineffizient.
Durch das Kopieren von Funktionsrümpfen an die Aufrufstellen (Expansion von Funktionsaufrufen), werden Funktionsaufrufe eingespart. Hierdurch kann das übersetzte Programm stark verbessert werden.

Expansion stellt Lokalität von Code aus verschiedenen Funktionen her. Dadurch können unter Umständen weitere gemeinsame Teilausdrücke mehrfach genutzt werden.

Beispiel:

```
f x = b x
g y = map (*2) (b y)
h z = append (f z) (g z)
```

In den Funktionen `f` und `g` wird eine Liste mit der Funktion `b` erzeugt. Die beiden Listen sind in $\mathcal{HDC}$ nicht direkt gemeinsam nutzbar, da für jede Funktion ein getrennter DAG (gerichteter azyklischer Graph) angelegt wird. Hier stehen gleiche Berechnungen in verschiedenen Funktionen und werden daher nicht entdeckt.

Eine Expansion der Funktionsaufrufe in der Funktion `h` ergibt:

```
h z = append (b z) (map (*2) (b z))
```

Der neue gemeinsame Teilausdruck `b z` kann nun mehrfach verwendet werden. Auf diese Weise wird die Berechnung einer unter Umständen sehr langen Liste eingespart. Die Funktion `h` wird dadurch beschleunigt und benötigt weniger Speicherplatz.

Zudem werden durch die Expansion von Funktionsaufrufen weitere Vereinfachungen möglich, z.B. die Auswertung von Operationen auf Konstanten zur Übersetzungszeit (*Konstantenfaltung*) oder das Löschen eines Zweiges bei `if-then-else`, wenn die Bedingung immer wahr oder immer falsch ist.

Schließlich können im Programm nicht mehr benötigte Funktionen aus dem Zwischencode entfernt werden.

1.2. Die Programmiersprache $\mathcal{HDC}$

$\mathcal{HDC}$ ($\mathcal{H}$igher-Order $\mathcal{D}$ivide-and-$\mathcal{C}$onquer) [HLG⁺99, HLLS99, Sch98] ist eine Teilsprache der rein funktionalen Programmiersprache Haskell. Sie ist strikt und erlaubt die statische Parallelisierung auf der Basis von Skeletten. Skelette sind Funktionen höherer Ordnung, für die auf der Zielarchitektur eine effiziente Implementierung existiert.

Wichtige Funktionen höherer Ordnung sind unter anderem `map` und `dc0`. `map` wendet eine Funktion auf alle Elemente einer Liste an. `dc0` [HL97a, HL97b] ist die allgemeinste Form von divide-and-conquer, dabei wird ein Problem rekursiv aufgeteilt, bis es trivial lösbar ist. Mit diesen Funktionen lassen sich eine große Anzahl von Algorithmen formulieren.

Der Übersetzungsvorgang des $\mathcal{HDC}$–Compilers ist in viele Phasen unterteilt [HLG+99]. Das Eingabeprogramm wird schrittweise durch Transformationen in ein C/MPI–Programm überführt. Das Message Passing Interface (MPI) [GLS94, For93] ist eine Bibliothek, die Funktionen zur Kommunikation zwischen Prozessoren oder Rechnern in verteilten Systemen bereitstellt.

Es werden die folgenden Phasen durchlaufen:

Scanner / Parser
: Es wird ein Syntaxbaum für das Programm aufgebaut. Der Parser wurde mit dem Parsergenerator `happy` aus einer Grammatik erstellt.

Vereinfachung von Mustervergleichen
: Geschachtelte Patterns werden aufgelöst. Muster in Funktionsargumenten werden in `case`–Ausdrücke auf der rechten Seite der Funktionsdefinition umgewandelt.

Vereinfachung von Listenkomprehensionen
: Mit einer Listenkomprehension (*list comprehension*) kann in Haskell der Inhalt einer Liste beschrieben werden. Man gibt eine Berechnungsvorschrift für die Werte der Elemente an. Sie werden aus den Elementen von Hilfslisten berechnet, die mit sogenannten Generatoren durchlaufen werden. Die Notation erinnert an die mathematische Mengenschreibweise. Die Berechnung von Listenkomprehensionen erfolgt unter Verwendung listenbasierter Funktionen für Konkatenation, elementweise Funktionsapplikation und Herausfiltern von Elementen. Die vordefinierten (Skelett–)Implementierungen dieser Funktionen ermöglichen eine parallele Berechnung.

Lambda Lifting, `let`–Elimination
: Ein Programm mit geschachtelten Funktionsdefinitionen wird in ein Programm umgewandelt, das nur noch aus Kombinatoren (globalen Funktionen) besteht. [Joh85]

Typ–Inferenz
: Die Typen für alle Ausdrücke werden berechnet und ihre Verträglichkeit geprüft.

Monomorphisierung
: Polymorphe Funktionen werden kopiert und die Typvariablen gemäß ihrer Verwendung spezialisiert.

Elimination funktionaler Argumente
: Funktionen, die als Argumente übergeben werden, werden durch Funktionscodes ersetzt. Dem Programm werden apply–Funktionen zur Auswertung der Codes hinzugefügt.

Elimination gegenseitiger Rekursion
Es sind zwei Verfahren implementiert, mit denen gegenseitige Rekursion in allen Programmen eliminiert werden kann. Diese Phase ist optional.

Elimination von `case`–Ausdrücken
Mustervergleiche werden in `if-then-else`–Abfragen umgewandelt.

Erzeugung von DAG–Zwischencode
Der Syntaxbaum jeder Funktion wird in einen gerichteten azyklischen Graphen umgewandelt. Auf der neuen Zwischencode–Ebene treten gemeinsame Teilausdrücke innerhalb einer Funktion nicht mehrfach auf.

Tupelelimination
Tupel werden in algebraische Datentypen umgewandelt. Die Tupelelimination vereinfacht die Speicherverwaltung bei der Codegenerierung. Sie wird schon an dieser Stelle durchgeführt, damit die weiteren Transformationen keine Sonderbehandlung für Tupel-Ausdrücke benötigen.

Optimierungsschleife
In mehreren Zyklen wird Expansion von Funktionsaufrufen verbunden mit Elimination gemeinsamer Teilausdrücke, Löschen von nicht erreichbarem Code, Vereinfachung von Ausdrücken und Größeninferenz durchgeführt.

Abstrakte Codegenerierung
Der abstrakte Code hat eine operationale Semantik. Die Ausdrücke liegen in einer Tabelle vor. Sie können in dieser Reihenfolge ausgewertet oder in eine geeignete Zielsprache übertragen werden.

Raum–Zeit–Abbildung
In dieser Phase wird (soweit möglich) der parallele Schedule für die Ausführung des Programms berechnet.

Codegenerierung
Übersetzt den abstrakten Code in den Zielcode. Derzeit wird C mit MPI–Aufrufen [GLS94, For93] verwendet.

1.3. Überblick

Diese Arbeit befaßt sich mit den Phasen, die auf die Elimination funktionaler Argumente folgen, bis einschließlich der abstrakten Codegenerierung. Die Tupelelimination und die Größeninferenz sind jedoch nicht eingeschlossen. Die Phasen sind in Abb. 1.1 dargestellt.

Zu Beginn wird die gegenseitige Rekursion eliminiert. Dieser Schritt kann auch ausgelassen werden. Kapitel 3 befaßt sich mit der Elimination gegenseitiger Rekursion.

Die Elimination von `case`–Ausdrücken (Kapitel 4) wird ebenfalls vor der Umwandlung des Zwischencodes in DAG–Darstellung durchgeführt, weil im DAG keine Mustervergleiche mehr vorgesehen sind. Diese Phase muß nach der Elimination der gegenseitigen Rekursion erfolgen, da dort unter Umständen `case`–Ausdrücke erzeugt werden.

In Kapitel 5 wird eine neue Zwischencode–Ebene eingeführt, in der auf der Basis von gerichteten azyklischen Graphen (DAGs) gemeinsame Teilausdrücke nur einmal berechnet werden müssen.

Für die weiteren Optimierungen wird eine Schleife mehrfach durchlaufen, in der Funktionsaufrufe expandiert und Vereinfachungsregeln angewendet werden (Kapitel 6).

Die abstrakte Codegenerierung verleiht dem DAG–Code eine operationale Semantik. Sie wird in Kapitel 7 beschrieben.

Eine Zusammenfassung der gewonnenen Erkenntnisse und ein Ausblick auf weitere sinnvolle Ergänzungen schließen die Arbeit ab.

1.4. Voraussetzungen

Für das Verständnis dieser Arbeit werden Grundkenntnisse der funktionalen Konzepte vorausgesetzt [Bir98, Rea89, Tho96]. Weiterhin ist Grundwissen im Bereich von Compilerbau [ASU86] hilfreich. In der Arbeit werden auch geläufige Begriffe aus der Graphentheorie verwendet [AU96, Nol76, Vol96].

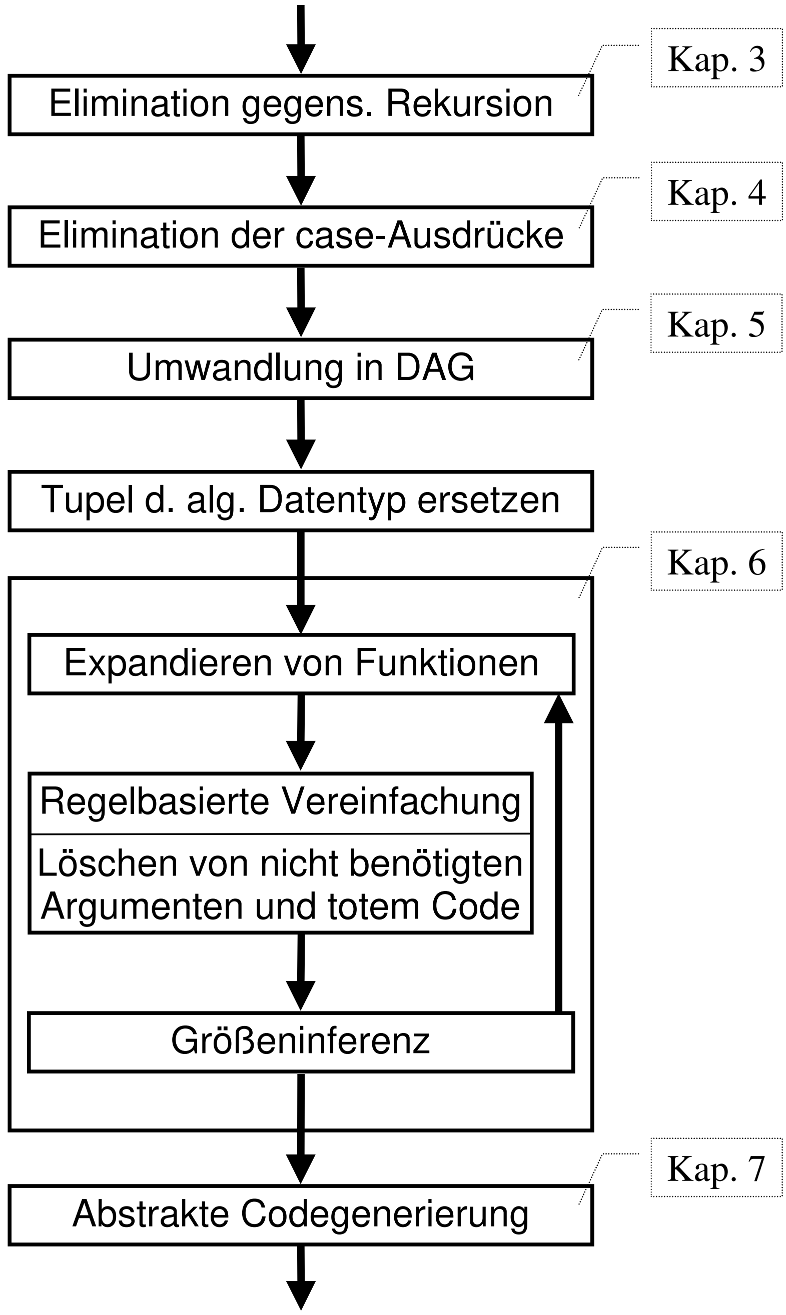

Abbildung 1.1.: Phasen vor Codegenerierung im Überblick

2. Grundlagen

2.1. Haskell

Haskell ist eine funktionale Programmiersprache, deren Bedeutung in letzter Zeit stark zunimmt. Eine Forschungsgruppe bei Microsoft (Cambridge) beschäftigt sich mit der Weiterentwicklung des Glasgow Haskell Compilers (GHC). Es wird auch daran gearbeitet, Haskell mit anderen Programmiersprachen zu verbinden und in Dokumente im World Wide Web zu integrieren.

Haskell ist *rein funktional* (keine Seiteneffekte möglich), *streng getypt*, *lazy* (nur benötigte Ausdrücke werden berechnet) und erlaubt Funktionen höherer Ordnung.

Eine Funktion, die eine andere Funktionen als Argument hat oder eine Funktion als Resultat liefert, wird als Funktion höherer Ordnung bezeichnet. In dem folgenden Beispiel wird eine Funktion in *Curry–Schreibweise* definiert. Wenn sie nicht beide Argument erhält (*partial application*), liefert sie wieder eine Funktion.

Beispiel:

```
add :: Int -> Int -> Int
add a b = a + b
```

Es sind nur die Werte von Ausdrücken wichtig, da keine Seiteneffekte auftreten können. Diese Eigenschaft wird als referentielle Transparenz (*referential transparency*) bezeichnet und ermöglicht die Umformung von Programmen durch Ersetzung von Gleichem mit Gleichem (*equational reasoning*). Es sind dafür keine aufwendigen Analysen nötig, wie sie für Sprachen, in denen Seiteneffekte möglich sind, gemacht werden müssen.

Das starke Typsystem hilft dabei, Fehler so früh wie möglich zu erkennen und zu vermeiden. Der Benutzer hat mit *algebraischen Datentypen* die Möglichkeit, eigene Typen zu definieren.

Es würde den Rahmen einer Diplomarbeit sprengen, eine allgemeine Einführung in Haskell zu geben. Das Tutorial „A Gentle Introduction to Haskell“ [HPF97] bietet einen schnellen Start. Es richtet sich an Leser, die bereits Kenntnisse in einer Programmiersprache haben. Für eine saubere und gründliche Einführung in die funktionale Programmierung ist [Bir98] empfehlenswert. Der Sprachumfang ist im „Haskell 98 Report“ [JH99a] definiert.

Spezielle Konstrukte (z.B. `case`–Ausdrücke) werden in dieser Arbeit erklärt, wo sie benötigt werden. An dieser Stelle sollen nur Arrays, algebraische Datentypen (mit „*labeled fields*") und die monadische Programmierung vorgestellt werden. Anstatt die Theorie dahinter zu erklären, werden nur Schreibweisen gezeigt. Sie werden benötigt, um in der Arbeit angegebene Funktionen und Algorithmen lesen zu können.

2.1.1. Arrays

Der Haskell Library Report [JH99b] macht folgende Aussage über Arrays:

> „Haskell provides indexable arrays, which may be thought of as functions whose domains are isomorphic to contiguous subsets of the integers. Functions restricted in this way can be implemented efficiently; in particular, a programmer may reasonably expect rapid access to the components."

Aufgrund der Allgemeinheit des Reports kann natürlich keine genauere Aussage über die Implementierung der verschiedenen Compiler gemacht werden. Der für diese Arbeit verwendete Glasgow Haskell Compiler unterstützt Zugriffe auf Arrays in $O(1)$–Zeit [JHH+93].

Für den schnellen Zugriff auf Elemente von Tabellen, wurden für die Implementierung der Codeoptimierung an mehreren Stellen Arrays eingesetzt. Obwohl das Anlegen eines Arrays eine teure Operation ist, kann eine Beschleunigung der Optimierung im Vergleich zur Implementierung mit AVL–Bäumen [AHU74] bestätigt werden.

Der Typ eines Arrays ist `Array a b`, wobei `a` ein Indextyp ist und `b` ein beliebiger Typ für die Elemente des Arrays. Folgende Funktionen werden verwendet:

```
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
bounds    :: (Ix a) => Array a b -> (a,a)
indices   :: (Ix a) => Array a b -> [a]
(!)       :: (Ix a) => Array a b -> a -> b
(//)      :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

Mit `listArray` wird aus einer Liste ein Array aufgebaut. Es werden dafür feste Indexgrenzen benötigt. `bounds` liefert die Grenzen eines Arrays. `indices` gibt alle Indices, die in einem Array verwendet werden, in einer Liste zurück. `(!)` liefert das Element an einer bestimmten Position, analog zu `(!!)` bei Listen. Elemente in einem Array können mit `(//)` verändert werden. Dazu wird eine Liste mit Positions-/Wertpaaren angegeben, die die neuen Werte enthält.

Beispiele:

```
a1 :: Array Int Int
a1 = listArray (1,10) [0,2..] -- erzeuge ein Array

a1 ! 5                        -- wähle das 5. Element aus

a2 = a1 // [(1,42)]           -- verändere 1. Element auf 42
```

2.1.2. Algebraische Datentypen

Algebraische Datentypen dienen zur Definition zusammengesetzter Datenstrukturen. Sie bestehen aus einer Menge von Konstruktoren, die Argumente mit Werten beliebiger Typen besitzen können.

Beispiel:
Ein Binärbaum mit inneren Knoten und Blättern, die Werte eines Typs a enthalten, wird wie folgt definiert:

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

Bezeichnete Felder (*labeled fields*) ermöglichen die Elemente eines Datentyps über Feldnamen anzusprechen. Die Feldnamen werden als Selektorfunktionen verwendet. Mit ihnen können, unabhängig von der Gesamtstruktur des Datentyps, Felder konstruiert, ausgewählt oder verändert werden.

Die Verwendung von Feldern ist hilfreich, wenn der Datentyp viele Komponenten enthält. Es ist dann für das Ändern oder Abfragen einer Komponente nicht nötig, einen langen Ausdruck für den Mustervergleich zu schreiben, in dem alle Komponenten aufgezählt werden. Zudem ist es einfacher, zu einem späteren Zeitpunkt Komponenten zu entfernen oder neue hinzuzufügen. Das Programm muß dazu nicht an jeder Stelle umgeschrieben werden, an der der Datentyp verwendet wird.

Beispiele:

```
data Color  = White | Gray | Black
data Fields = Fields { color::Color, nr::Int }

f1 = Fields { color=White, nr=0 } -- konstruiere Fields
c  = color f1                     -- ermittle Farbe von f1
f2 = f1 { nr=nr f1 + 1 }          -- erhöhe den Wert von nr
```

2.1.3. Monaden

Viele effiziente „imperative“ Algorithmen verändern globale Variablen. Solche Seiteneffekte sind in rein funktionalen Sprachen unerwünscht. Leider gibt es oft keine ebenso effiziente funktionale Spezifikation für diese Algorithmen. Das mathematische Konzept der Monade ermöglicht eine Beschreibung von Berechnungen (*computations*) [Mog91, Wad92]. Mit einer Monade kann ein Zustand gekapselt werden, der bei der Komposition von Berechnungen durch die Bindeoperation versteckt weitergegeben wird.

Eine Monade wird definiert mit einem Typkonstruktor `M` und den polymorphen Funktionen

```
return :: a -> M a
bind   :: M a -> (a -> M b) -> M b
```

`M a` ist der Typ einer Berechnung mit dem Resultat `a`. Programme können als mit `bind` verkettete Berechnungen geschrieben werden. Dabei ist jeweils der nächste Schritt eine Funktion, die das Resultat des vorhergehenden Schritts übernimmt.

Ein derartiges Programm ist durch die Verwendung vieler Funktionen höherer Ordnung nur schwer lesbar. Die do–Notation ist syntaktischer Zucker (*syntactic sugar*) für die Verkettungen. Es können damit Berechnungen ähnlich wie in einer imperativen Sprache untereinander geschrieben werden. Die Ergebnisse werden mit dem umgekehrten Pfeil (`<-`) an einen Namen gebunden. Dabei erzeugt jede Zeile einen neuen Bindungsbereich (*scope*).

Beispiel:
Die `IO`–Monade stellt unter anderem „Berechnungen“ für die Ein-/Ausgabe von Zeichenketten zur Verfügung:

```
getLine :: IO String
putStr :: String -> IO ()
```

Das folgende Beispielprogramm verwendet die do–Notation:

```
import IO

main = do putStr "Bitte geben Sie ihren Namen ein: "
          name <- getLine
          putStr ("Hallo "++name)
```

Um eine Monade verwenden zu können, müssen Implementierungen für `return` und `bind` angegeben werden. In der Implementierung von $\mathcal{HDC}$ liegt eine geeignete Definition vor.

2.2. Darstellung von Algorithmen

Die meisten der verwendeten Algorithmen werden in dieser Diplomarbeit funktional mit Angabe von Typen spezifiziert. Dazu werden auch Funktionen verwendet, die in der *Standard Prelude* von Haskell [JH99a] definiert sind. Darüber hinaus werden die in den Abschnitten 2.1.1 bis 2.1.3 erklärten Schreibweisen verwendet.

In den algorithmischen Definitionen werden zur Vereinfachung zum Teil auch Berechnungen oder Bedingungen in intuitiv verständlichem Text formuliert.

2.3. Aufrufgraph

Definition 2.3.1 (*Aufrufgraph*)
Sei $\mathcal{V} \subset$ `Int` die Menge der Funktionsindices aller im Zwischencode definierten Funktionen. Ein gerichteter Graph $CG = (V, E)$, $V \subseteq \mathcal{V}$, wird als Aufrufgraph für ein Programm bezeichnet, falls die Indices aller benötigten Funktionen in V enthalten sind und es eine Kante $(f_1, f_2) \in E$ gibt gdw. Funktion mit Index f_1 enthält mindestens einen Funktionsaufruf der Funktion mit Index f_2.

Der Aufrufgraph läßt sich nach der Elimination funktionaler Argumente immer statisch bestimmen, da es dann nicht mehr möglich ist, daß eine Funktion an eine andere übergeben wird.

Mehrfache Kanten zwischen zwei Knoten sind aus der Definition ausgenommen, weil es für die weiteren Betrachtungen keine Rolle spielt, ob eine Funktion nur einmal oder mehrfach innerhalb eines Funktionsrumpfs aufgerufen wird. Der Aufrufgraph wird daher in beiden Fällen für die Repräsentation der Abhängigkeit nur eine Kante enthalten.

Beispiel:
Abb. 2.1 zeigt ein abstraktes Programm mit dazu gehörigem Aufrufgraph.

```
parmain = ... f1() ...
f1 = ... f2() ... f4() ...
f2 = ... f3() ...
f3 = ... f1() ...
f4 = ... f3() ...
```

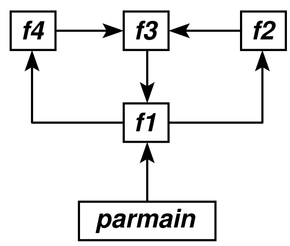

Abbildung 2.1.: Beispiel für einen Aufrufgraph

In der Zwischencoderepräsentation ist jede Funktion unter einem eindeutigen Index abgelegt. Das Programm benötigt aber in der Regel nicht alle Funktionen, und eine Funktion verwendet ihrerseits nur einen kleinen Teil der Funktionen des Programms. So erweist es sich als sinnvoll, den Aufrufgraph in einer Adjazenzliste abzuspeichern:

```
type CallGraph = [(Int,[Int])]
```

Hierbei wird für den Funktionsindex jeder benötigten Funktion eine Liste mit den Funktionsindices der im Rumpf verwendeten Funktionen abgelegt.

2.4. Tiefensuche

Die Tiefensuche wird verwendet, um einen Graphen so zu durchlaufen, daß jeder Knoten genau einmal besucht wird. Als Eingabe wird der Graph und eine Reihenfolge seiner Knoten benötigt.

Vorgehensweise: Die Knoten werden in der angegebenen Reihenfolge nacheinander als Startknoten für die Suche verwendet. Dadurch kann die Tiefensuche auch angewendet werden, wenn der Graph nicht eine Wurzel hat, von der alle Knoten erreichbar sind. Falls ein Knoten bereits gefunden wurde, wird dieser nicht weiter beachtet. Ansonsten wird die Suche bei seinen Nachfolgerknoten rekursiv fortgesetzt.

Das Ergebnis der Tiefensuche ist eine Liste. Sie enthält für jeden Startknoten s eine Liste l_s der Knoten, die von s aus zum ersten Mal gefunden wurden. l_s ist leer, falls s bereits ausgehend von einem vorangegangenen Startknoten gefunden wurde. Jeder Knoten des Graphen tritt in genau einer Liste im Ergebnis auf.

```
type M a = State DFSState a
type Graph = Array Int [Int]

data Color    = White | Gray | Black
data DFSState = DFSState { v :: [Int],
                           c :: Array Int Color }

dfs :: Graph -> [Int] -> M [[Int]]
dfs graph nodes
  = do updateState (\s -> s { v = [],
                              c = listArray (0,maximum nodes)
                                            (repeat White) })
       mapM (dfsvisit graph) nodes

dfsvisit :: Graph -> Int -> M [Int]
dfsvisit graph node
  = do state <- fetchState
       if c state ! node == White
       then do
         updateState (\s -> s { c = c s // [(node,Gray)] })
         found <- mapM (dfsvisit graph) (graph ! node)
         updateState (\s -> s { c = c s // [(node,Black)],
                                v = node : v s })
         return (node : concat found)
       else return []
```

Abbildung 2.2.: Funktion für die Tiefensuche in einem Graphen

Aus Effizienzgründen wird bei der Tiefensuche der in ein Array umgewandelte Aufrufgraph verwendet. Das Ermitteln der von einem Knoten ausgehenden Kanten kann damit in konstanter Zeit erfolgen, eine geeignete Übersetzung durch den Haskell–Compiler vorausgesetzt. In der Listenrepräsentation ist es nur in linearer Zeit möglich, weil die Liste durchsucht werden muß.

Die Funktion `dfs` (siehe Abb. 2.2) liefert das Ergebnis der Tiefensuche. Zu Beginn werden alle Knoten weiß gefärbt. Dann wird `dfsvisit` für jeden Knoten in der Reihenfolge `nodes` aufgerufen. `dfsvisit` prüft zuerst, ob der Knoten noch nicht gefunden wurde (Farbe weiß). Ist das der Fall, so wird er bearbeitet: Er wird grau gefärbt und die Funktion rekursiv auf seine Nachfolgerknoten angewendet. Beim Verlassen des Knotens wird er schwarz gefärbt. Andernfalls konnte `dfsvisit` keinen neuen Knoten finden und liefert die leere Liste.

Das Ergebnis der Tiefensuche hängt von der Reihenfolge der Startknoten, die hier durch die Position in der Liste `nodes` gegeben ist, ab.

Für die Suche nach starken Zusammenhangskomponenten wird die umgekehrte Postorder–Reihenfolge (UPR) der Knoten aus der Tiefensuche benötigt. In dieser Reihenfolge sind die Knoten wie folgt geordnet: Wenn die rekursive Suche von Knoten k_1 aus *nach* der rekursiven Suche ausgehend von Knoten k_2 beendet ist, so ist $k_1 < k_2$. D.h. k_1 ist in der Reihenfolge vor k_2 angesiedelt.

Für die Erzeugung der UPR könnten die Knoten als letzte Aktion in `dfsvisit` – also dann, wenn der Knoten schwarz gefärbt wird – mit einer fortlaufenden Nummer versehen werden. Die Knoten müßten im Anschluß an die Tiefensuche nach absteigenden Nummern sortiert werden (siehe [CLR90]).

Einfacher ist jedoch, die Knoten immer am Anfang einer Liste einzufügen. Diese Vorgehensweise wurde in Abb. 2.2 gewählt. Die so aufgebaute Liste `v` enthält nach Abschluß der Tiefensuche die Knoten in UPR.

Beispiel:
Abb. 2.3 zeigt anhand eines Beispielgraphen schrittweise den Ablauf der Tiefensuche. Dazu ist derselbe Graph 15 mal abgebildet. Die Nummerierung der Knoten ist in Abb. 2.3(A) angegeben und gilt ebenso für die weiteren Abbildungen des Graphen. Die Array–Darstellung des Graphen wird erzeugt mit:

```
graph = listArray (1,7) [[2,3],[4],[4,5],[1],[],[],[6]]
```

`dfs graph [1..7]` liefert folgendes Ergebnis der monadischen Berechnung:

```
[[1,2,4,3,5],[],[],[],[],[6],[7]]
```

Im folgenden wird betrachtet, wie die Tiefensuche in Abb. 2.3 schrittweise abläuft.

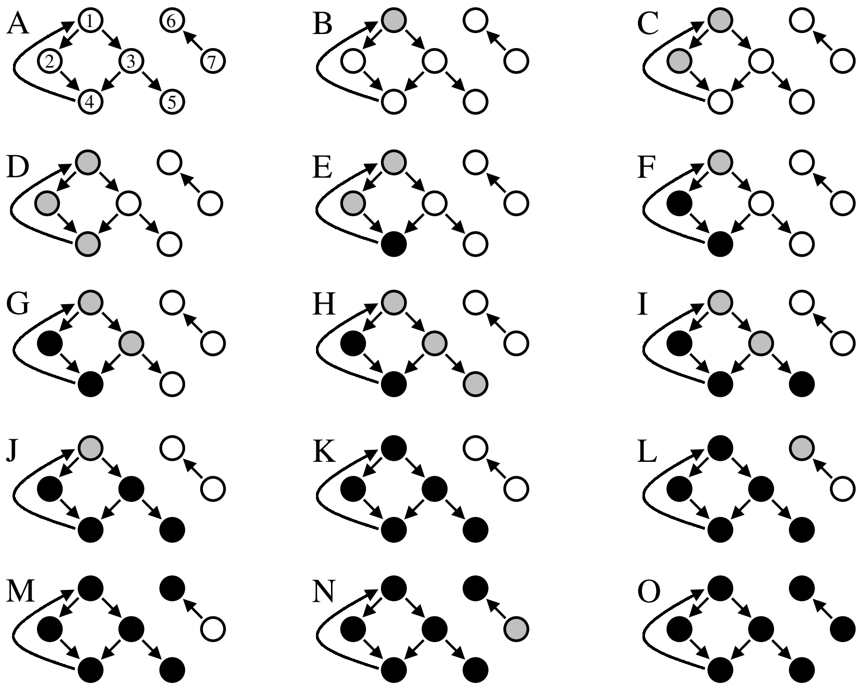

Abbildung 2.3.: Beispiel: Tiefensuche mit Knotenfärbung

Die Suche beginnt mit Knoten 1. Von hier werden 2, dann 4 gefunden. Jeder neu gefundene Knoten wird grau gefärbt. Von Knoten 4 wird wieder 1 getestet. Da 1 nicht mehr weiß ist und 4 keine anderen Nachfolger hat, ist Knoten 4 vollständig bearbeitet. Er wird deshalb schwarz gefärbt (Abb. 2.3(E)). Die Rekursion geht auf der vorigen Ebene weiter, jedoch auch von Knoten 2 aus sind keine weiteren Kanten zu verfolgen. Es wird dann die zweite Kante von Knoten 1 betrachtet und Knoten 3 gefunden (Abb. 2.3(G)). Hier existiert eine Kante zu Knoten 4, der nicht mehr weiß ist. Dieser Knoten wird deswegen nicht weiter betrachtet und es wird Knoten 5 gefunden. Das Ende aller Rekursionen vom Startknoten 1 ist erreicht. Die Suche für die Startknoten 2..5 liefert nur leere Listen im Ergebnis, weil diese Knoten bereits gefunden wurden. Knoten 6 als Startknoten liefert nur `[6]` weil er keine Nachfolger hat. Von Knoten 7 werden außer 7 selbst keine neuen Knoten mehr gefunden. Damit ist die gesamte Tiefensuche abgeschlossen.

Im Beispiel werden in den Bildern E, F, I, J, K, M, O die Knoten 4, 2, 5, 3, 1, 6, 7 schwarz gefärbt. Da sie immer an den Anfang der Liste `v` gesetzt werden, enthält `v` nach der Tiefensuche die UPR `[7,6,1,3,5,2,4]`.

2.5. Starke Zusammenhangskomponenten

Die starken Zusammenhangskomponenten werden hier verwendet, um die Funktionen in Bereiche einzuteilen, in denen gegenseitige Rekursion auftritt.

Definition 2.5.1 (*starke Zusammenhangskomponente*)
Sei $G = (V, E)$ ein gerichteter Graph. V kann in r Äquivalenzklassen $V_i, 1 \leq i \leq r$, eingeteilt werden: die Knoten v und w seien äquivalent gdw. es gibt einen Weg von v nach w und einen Weg von w nach v. Sei E_i, $1 \leq i \leq r$, die Menge von Kanten, deren Start- und Endknoten in V_i liegen. Die Graphen $G_i = (V_i, E_i)$ werden *starke Zusammenhangskomponenten* genannt. [AHU87]

Für die Suche nach starken Zusammenhangskomponenten habe ich den Algorithmus aus [CLR90, S. 489] implementiert, dessen Korrektheit dort auch bewiesen ist. Der Algorithmus führt eine Tiefensuche auf dem Graphen aus, transponiert den Graphen, d.h. es werden alle Kanten umgedreht, und führt schließlich eine weitere Tiefensuche auf dem transponierten Graphen durch. Bei der zweiten Tiefensuche müssen für diesen Algorithmus die Knoten in der umgekehrten Postorder–Reihenfolge (siehe Abschnitt 2.4) betrachtet werden. Diese Reihenfolge wurde in der Zustandsvariable `v` abgelegt. `fetchState` gibt den aktuellen Zustand zurück. Es kann damit der Wert der Zustandsvariable ermittelt werden.

```
search_sccs :: Graph -> M [[Int]]
search_sccs graph
  = do dfs graph (indices graph)
       state  <- fetchState
       tgraph <- transposeGraph graph
       sccs   <- dfs tgraph (v state)
       return (filter (not . null) sccs)
```

Abbildung 2.4.: Berechnung der starken Zusammenhangskomponenten

Die Zeitkomplexität ist in $O(n^2)$, wobei n die Anzahl der Knoten im Aufrufgraphen ist. Eine Tiefensuche ruft `dfsvisit` für alle Knoten auf. Für jedem Knoten wird sie, sofern dieser Knoten noch nicht besucht wurde, für alle Nachfolgerknoten aufgerufen. Die Anzahl der Nachfolger kann aufgrund der in Abschnitt 2.3 definierten Datenstruktur des Aufrufgraphen maximal n sein. Daraus ergibt sich die Komplexität $O(n^2)$ für eine Tiefensuche. Das Umdrehen einer Kante muß für alle Kanten durchgeführt werden. Da es im schlimmsten Fall n^2 Kanten gibt, bleibt die Komplexität des Algorithmus in $O(n^2)$. (Für diese Betrachtung wurde angenommen, daß der Zugriff auf ein Array–Element in konstanter Zeit erfolgt.)

King und Launchbury haben sich auch mit der Tiefensuche und dem Auffinden von starken Zusammenhangskomponenten beschäftigt [KL95]. Sie leiten ein Haskell–Programm her, das aus vielen, aber übersichtlichen Funktionen besteht und monadische Elemente enthält. Es implementiert ebenfalls den in dieser Arbeit verwendeten Algorithmus zur Bestimmung starker Zusammenhangskomponenten. In diesem Papier wird die Korrektheit auf eine andere Art bewiesen als in [CLR90]. Leider wurden *updatable arrays* verwendet, die nicht konform mit dem Haskell–Interpreter *Hugs* waren. Deswegen habe ich mich für die eigene Lösung mit Hilfe der im Projekt bereits vorhandenen Definition einer Zustandsmonade entschieden.

2.6. Meßwerte

Bewertung und Vergleich verschiedener Optimierungen erfolgen durch Messungen, die mit dem $\mathcal{HDC}$–Interpreter durchgeführt werden können. Interessant sind folgende Werte (die in Tabellen verwendeten Abkürzungen sind in Klammern angegeben):

Operationen (Op)
: Die Anzahl der zur Berechnung des Ergebnisses eines Programms durchgeführten Operationen.

Schedule–Zeit (Sched)
: Ein primitiver Scheduler liefert eine Abschätzung, wieviele Schritte bei Ausführung auf einem Parallelrechner benötigt werden. Es wird für die Messungen hier angenommen, daß eine unbegrenzte Anzahl von Prozessoren zur Verfügung steht, also jede Operation zum frühestmöglichen Zeitpunkt ausgeführt werden kann (*free schedule*). Von den Kosten für Kommunikationen wurde abstrahiert.

Durchschnittlicher Parallelitätsgrad (Par)
: Die Anzahl der Prozessoren, die im Schnitt für die Berechnung benötigt wurden.

Für den Vergleich von Programmgrößen werden folgende Werte bestimmt:

Anzahl von Funktionen (Fkt)
: Es werden alle Funktionen gezählt, die benötigt werden und für die im Zwischencode (Syntaxbaum bzw. DAG) eine Definition existiert.

Anzahl von Knoten (Kn)
: Gibt an, wieviele Syntaxbaum– bzw. DAG–Knoten das Programm in den Definitionen der benötigten Funktionen enthält.

3. Elimination gegenseitiger Rekursion

3.1. Motivation

Viele Optimierungen werden erschwert, wenn sich die Funktionen eines Programms gegenseitig rekursiv aufrufen. Es ist dann z.B. schwieriger zu erkennen, bei welchen Funktionen Argumente eliminiert werden können, weil sie keinen Beitrag zur Berechnung liefern.

Indirekte Rekursionen sind auch ungünstig für die Phase der Größeninferenz, in der die Typen durch symbolische Informationen über die Größe der Objekte versehen werden.

Im folgenden werden zwei Verfahren vorgestellt, mit denen die gegenseitige Rekursion (*mutual recursion*) innerhalb von starken Zusammenhangskomponenten eliminiert werden kann. Das erste Verfahren ist nicht allgemein anwendbar. Falls die Voraussetzungen erfüllt sind, wird jedoch in der Regel besserer Code generiert, der auch bei der Größeninferenz besser analysiert werden kann.

3.2. Elimination durch Expansion von Funktionen

Satz 3.2.1 (*Anwendbarkeit der Elimination durch Expansion*)
Für eine starke Zusammenhangskomponente existiert eine Folge von Expansionen zur Elimination gegenseitiger Rekursion, falls alle Zyklen innerhalb der starken Zusammenhangskomponente einen gemeinsamen Knoten haben.

Beweis 3.2.1
Annahme: Alle Zyklen innerhalb einer starken Zusammenhangskomponente G laufen durch einen gemeinsamen Knoten k. Wird k aus G gelöscht, erhält man einen gerichteten azyklischen Graphen H. Sonst gäbe es einen Zyklus, der nicht durch k geht. Es ist damit möglich, H vollständig in k zu expandieren: Solange es Funktionsaufrufe von Funktionen aus G in k gibt, ersetze sie durch ihren Rumpf. Da H keine Zyklen enthält, ist die Termination gewährleistet. Es wird ein Endzustand erreicht, in dem alle Zyklen von G in direkt rekursive Aufrufe in k umgewandelt sind.

Die Idee für dieses Verfahren stammt aus dem Beweis eines Theorems in [KRP93, Seite 7]. Das Theorem selbst ist umstritten (siehe [YK97]) und wird in dieser Arbeit nicht verwendet.

Beispiel:
In Abb. 3.1(A) ist ein Aufrufgraph einer starken Zusammenhangskomponente dargestellt. Die Knoten stehen für Funktionen, die Kanten für Funktionsaufrufe. Man erkennt zwei Zyklen: 1–2–4–3–1 und 2–4–3–2. Die Knoten 2 und 3 liegen auf beiden Zyklen. Für die Elimination wurde Knoten 3 gewählt. Die Abbildungen B bis F zeigen, wie sich der Graph verändert, wenn die Aufrufe in Knoten 3 nacheinander expandiert werden: Durch eine Expansion fällt der gestrichelte Pfeil weg und es kommen Aufrufe des kopierten Funktionsrumpfes hinzu (dicke Pfeile). Die Expansion wird wiederholt bis es schließlich keine gegenseitige Rekursion mehr gibt. Die Funktionen 1, 2, 4 dürfen nicht gelöscht werden, falls sie im Programm noch benötigt werden, da die veränderte Funktion nicht als Ersatz für die gesamte starke Zusammenhangskomponente dient. Ein Aufruf einer dieser Funktionen wird immer, falls der Rekursionsfall eintritt, in der jetzt direkt rekursiven Funktion 3 enden.

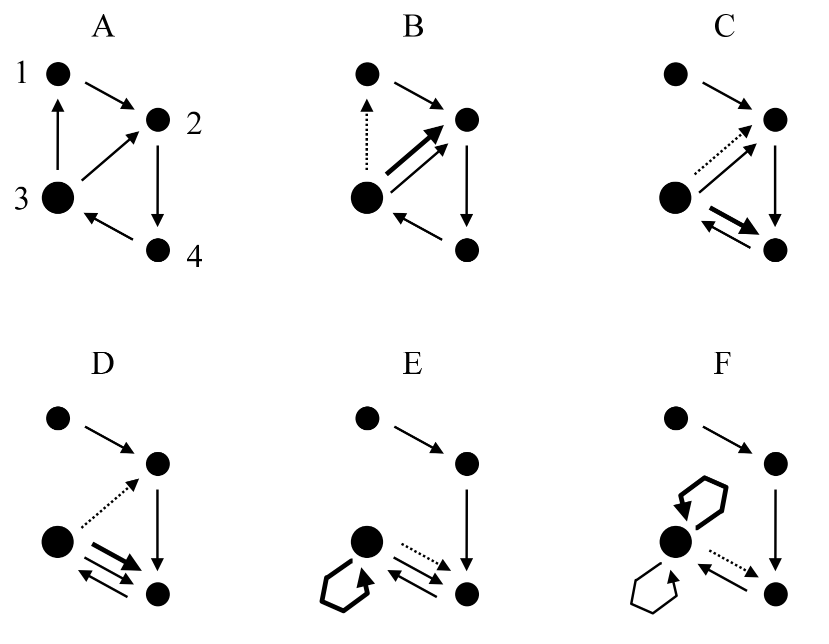

Abbildung 3.1.: Beispiel: Elimination gegenseitiger Rekursion durch Expansion

3.3. Elimination durch Zusammenfassen von Funktionen

Indirekte Rekursion kann immer aufgelöst werden, indem man für jede Menge gegenseitig rekursiver Funktionen f_i eine neue Funktion f' einführt, die alle Funktionen der jeweiligen Menge zusammenfaßt. So wird die gegenseitige Rekursion zu direkter Rekursion in f' reduziert. [KRP93, Einleitung]

Das Verfahren zur Konstruktion der Funktion f' soll im allgemeinen Fall anwendbar sein. Insbesondere ergeben sich Schwierigkeiten, wenn sich die Typsignaturen der Funktionen f_i unterscheiden, bzw. an einer bestimmten Argumentposition verschiedene Typen übergeben werden. Eine mögliche Lösung verwendet einen algebraischen Datentyp für das Ergebnis und die Komponenten des Arguments von f'.

Im folgenden wird in drei Schritten die gegenseitige Rekursion in einer starken Zusammenhangskomponente eliminiert:

1. Erzeugen der neuen Funktion f', die aus den Rümpfen aller Funktionen der starken Zusammenhangskomponente besteht,
2. Ersetzen der alten Funktionsrümpfe durch Aufruf der neuen Funktion,
3. Expansion von Aufrufen alter Funktionen in der neuen Funktion.

Gegeben seien n Funktionen f_i, $i \in \{1 \ldots n\}$. Die Anzahl der Argumente von Funktion f_i sei $m(i)$.

$$
\begin{array}{l}
f_i :: t_{(i,1)} \texttt{ -> } t_{(i,2)} \texttt{ -> } \ldots \texttt{ -> } t_{(i,m(i))} \texttt{ -> } t_{(i,0)} \\
f_i \;\; arg_{(i,1)} \;\; arg_{(i,2)} \;\; \ldots \;\; arg_{(i,m(i))} \;\texttt{=}\; Rumpf_i
\end{array}
$$

Die Funktionen werden zusammengefaßt zu einer neuen Funktion f'. Ihr Argument wird als Funktionscode verwendet. Es kodiert die ursprüngliche Funktion aus der starken Zusammenhangskomponente und deren Argumente. In f' werden alle Funktionsrümpfe der Funktionen f_i kopiert. Der jeweils benötigte Rumpf wird durch die Dekodierung des Funktionscodes ermittelt, dabei werden auch die Argumente dekodiert.

$$
\begin{array}{l}
f' :: \texttt{Data -> Data} \\
f' \;\; arg \\
\quad \texttt{= case } arg \texttt{ of} \\
\qquad\quad C_1 \;\; arg_{(1,1)} \;\; \ldots \;\; arg_{(1,m(1))} \quad \texttt{->} \; CR_{t(1,0)} \;\; Rumpf_1 \\
\qquad\quad \vdots \\
\qquad\quad C_n \;\; arg_{(n,1)} \;\; \ldots \;\; arg_{(n,m(n))} \quad \texttt{->} \; CR_{t(n,0)} \;\; Rumpf_n
\end{array}
$$

Es muß für jede Funktion f_i ein Konstruktor C_i erzeugt werden, mit dem die Argumente von f_i und die alte Funktionsnummer kodiert werden. Weiterhin muß für jeden Typ $t_{(i,0)}$ des Funktionsergebnisses der Funktion f_i ein Konstruktor

$CR_{t(i,0)}$ erzeugt werden. Die Konstruktoren haben die Argumente bzw. das Ergebnis einer Funktion aus der ehemals starken Zusammenhangskomponente als Parameter und liefern einen Wert vom Typ des algebraischen Datentyps `Data`.

Die neu erzeugte Funktion f' kann jede der in ihr zusammengefaßten Funktionen durch einen geeigneten Aufruf auswerten. Somit können nun die alten Funktionsrümpfe der Funktionen f_i ersetzt werden durch einen Aufruf der neuen Funktion:

$$
\begin{array}{l}
f_i :: t_{(i,1)} \texttt{ -> } t_{(i,2)} \texttt{ -> } \ldots \texttt{ -> } t_{(i,m(i))} \texttt{ -> } t_{(i,0)} \\
f_i \;\; arg_{(i,1)} \;\; arg_{(i,2)} \;\; \ldots \;\; arg_{(i,m(i))} \\
\quad \texttt{= case } f' \; \texttt{(} C_i \;\; arg_{(i,1)} \;\; \ldots \;\; arg_{(i,m(i))} \texttt{) of} \\
\qquad\quad CR_{t(i,0)} \;\; result \texttt{ -> } result
\end{array}
$$

Im Programm kann eine Funktion f_i wie bisher weiter verwendet werden. Es ist also nicht nötig, Funktionsaufrufe von f_i zu verändern.

Das Programm enthält immer noch gegenseitige Rekursionen: die in f' kopierten Funktionsrümpfe enthalten noch Aufrufe von Funktionen f_i in der starken Zusammenhangskomponente. Der Rumpf einer Funktion f_i besteht jedoch nur noch aus einem Aufruf von f'. Diese Indirektion läßt sich durch Expansion der Funktionsaufrufe eliminieren. Es verbleiben damit nur direkt rekursive Funktionsaufrufe und Aufrufe von Funktionen in anderen starken Zusammenhangskomponenten im Rumpf von f'.

Beispiel:
In Abb. 3.2 wird das Verfahren mit einem Beispiel verdeutlicht. Es wird von derselben starken Zusammenhangskomponente wie im Beispiel des letzten Abschnitts ausgegangen. Abb. 3.2(B) zeigt den ersten Schritt: Die neue Funktion enthält Kopien der Funktionsrümpfe der Funktionen der starken Zusammenhangskomponente. Sie enthalten also gleiche Funktionsaufrufe wie die Originalfunktionen. Nach dem zweiten Schritt (Abb. 3.2(C)) sind die alten Funktionsrümpfe modifiziert. Sie rufen jetzt nur noch die neue Funktion auf. Durch den letzten Schritt wird schließlich die gegenseitige Rekursion aufgebrochen.

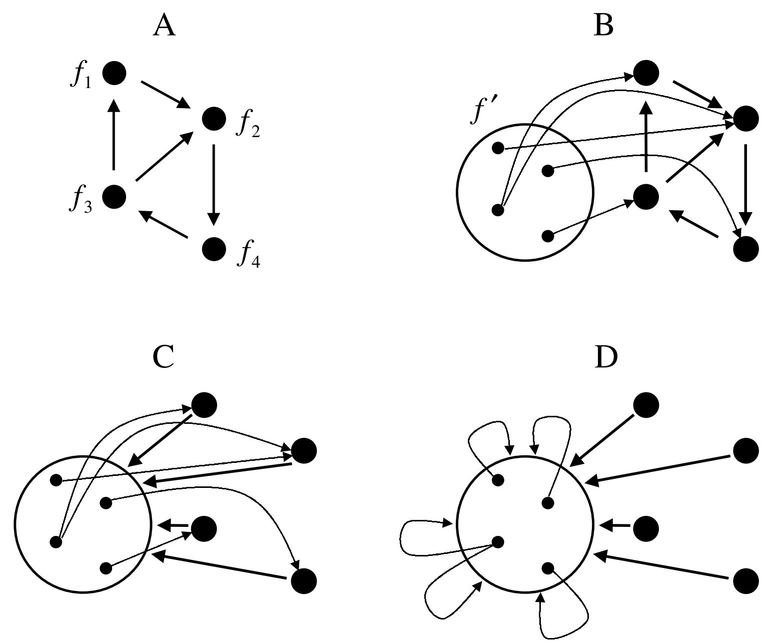

Abbildung 3.2.: Beispiel: Elimination gegenseitiger Rekursion durch Zusammenfassen

Anmerkung: Nach jedem Schritt liegt ein typkorrektes Programm mit unveränderter Semantik vor. Die gegenseitige Rekursion in einer starken Zusammenhangskomponente ist nach dem dritten Schritt aufgelöst.

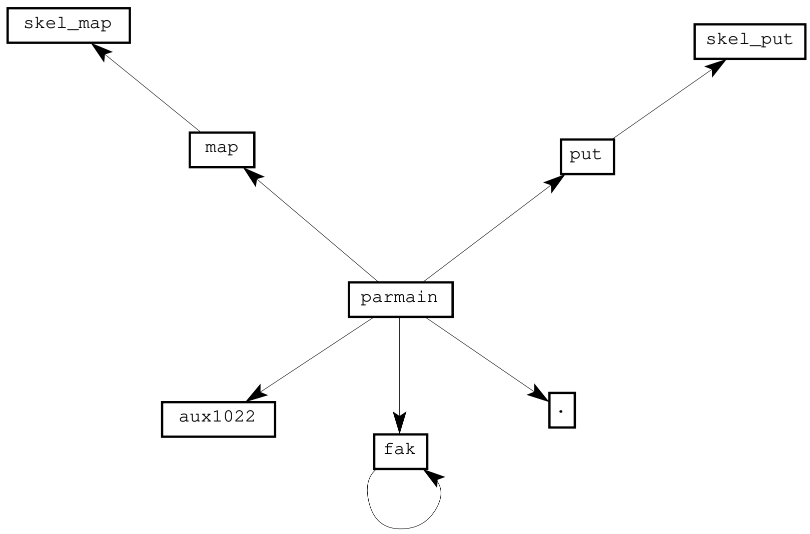

Abbildung 3.3.: Beispiel: Aufrufgraph vor Elimination funktionaler Argumente

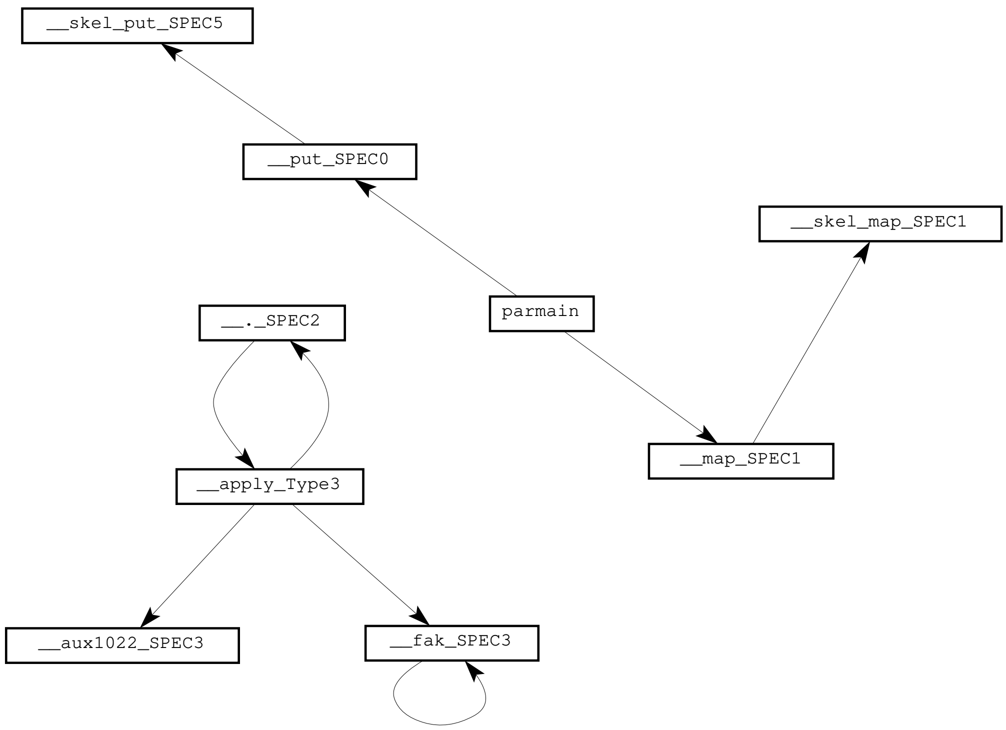

Abbildung 3.4.: Beispiel: Aufrufgraph nach Elimination funktionaler Argumente

```
fak :: Int -> Int
fak 0 = 1
fak a = a * fak (a-1)

parmain :: IO Unit
parmain = put (map ((+1) . fak) [1..10])
```

Abbildung 3.5.: Beispielprogramm: Entstehung gegenseitiger Rekursion

3.4. Auswirkungen auf den Code

Beide vorgestellten Methoden sind teuer, wenn das Programm starke Zusammenhangskomponenten mit vielen Knoten enthält. Deswegen kann diese Phase durch die Compileroptionen ausgeschaltet werden, falls der erzeugte Code durch die Elimination zu ineffizient werden sollte.

Zyklen im Programm können auch durch die vorangegangenen Transformationen entstehen. Jedoch bleibt ihre Anzahl klein, wenn der Benutzer Skelette anstelle von eigenen rekursiven Definitionen verwendet.

Wie das folgende Beispiel zeigt, ist es möglich, daß bei der Transformation zur Elimination funktionaler Argumente [Sch98] ein Zyklus erzeugt wird. In den Bildern 3.3 und 3.4 ist der Aufrufgraph des Beispielprogramms (Abb. 3.5) vor und nach der Transformation dargestellt. Man erkennt, daß sich die dabei neu erzeugte Funktion `__apply_Type3` und die spezialisierte Funktion für Funktionskomposition (`__._SPEC2`) gegenseitig aufrufen.

Der Aufrufgraph ist nach der Transformation nicht mehr zusammenhängend. Es werden trotzdem alle Funktionen benötigt, weil `__apply_Type3` vom Code des `map`–Skeletts aufgerufen wird. Die Funktion `aux1022` ist durch das Lambda–Lifting für die partielle Applikation `(+1)` erzeugt worden.

Ein Vergleich der $\mathcal{HDC}$–Beispielprogramme `freqset.hdc` und `freqset_rek.hdc` (siehe Anhang) zeigt das deutliche Effizienzproblem bei der Elimination gegenseitiger Rekursion in Programmen, die viele rekursive Funktionen enthalten. Dadurch besteht die Möglichkeit, daß die Elimination funktionaler Argumente große starke Zusammenhangskomponenten erzeugt. Für `freqset.hdc` wurden Skelette verwendet. `freqset_rek.hdc` verwendet Skelette nur für die Ein-/Ausgabe. Alle anderen Skelette wurden durch rekursive Definitionen ersetzt. Sie ähneln den Definitionen der *Haskell Standard Prelude* [JH99a, Anhang]. Wird ein Programm durch Graphreduktion ausgewertet, sind die rekursiven Definitionen unproblematisch. In $\mathcal{HDC}$ sollten jedoch die vordefinierten

Skelette bevorzugt verwendet werden: Das Programm kann besser parallelisiert werden und die Wahrscheinlichkeit, daß große starke Zusammenhangskomponenten durch die Elimination funktionaler Argumente erzeugt werden, ist geringer.

Tabelle 3.6 stellt die beiden Programme gegenüber.

Ein Problem sind die großen starken Zusammenhangskomponenten mit 4 und 21 Funktionen. Sie können nicht durch Expansion aufgelöst werden. Das Zusammenfassen der Funktionen in diesen starken Zusammenhangskomponenten erzeugt große Funktionen. Der abstrakte Code von `freqset_rek` wächst bei Elimination der gegenseitigen Rekursion um ca. 18%. Dies liegt jedoch unter Umständen zum Teil auch daran, daß auf Grund des veränderten Codes die Transformationen der Optimierung nicht an den selben Stellen angewendet werden können.

Für die Messung der Anzahl der Operationen wurde eine kleine (unrealistische) Eingabe verwendet, da Ausführungszeit des Interpreters bei `freqset_rek` mit Elimination gegenseitiger Rekursion sehr lang werden kann.

Wie zu erwarten, ist die Anzahl der Operationen bei der rekursiven Version größer als bei der Skelettversion. Wird die gegenseitiger Rekursion eliminiert, vergrößert sich diese Anzahl nochmals um den Faktor 5. Dieser Overhead entsteht durch die erzeugten Konstruktoren und Mustervergleiche. Das Programm wird bei diesem Beispiel so stark durch die Transformation verändert, daß es nicht mehr effizient ausgeführt werden kann.

	`freqset`		`freqset_rek`	
Programm vor Optimierung				
Anzahl der Funktionen	103		115	
Syntaxbaumknoten	968		1636	
Starke Zusammenhangskomponenten				
mit 2 Funktionen	0		10	
mit 3 Funktionen	1		0	
mit 4 Funktionen	0		1	
mit 21 Funktionen	0		1	
Elimination g. Rekursion	Ja	Nein	Ja	Nein
Programm nach Optimierung				
Anzahl der Funktionen	27	27	46	51
Anzahl der Knoten im DAG	465	412	1192	1152
Anzahl der Knoten im abstr. Code	573	493	1741	1474
Interpreter Operationen	2289	2289	15940	3187

Abbildung 3.6.: Beispiel: Auswirkung der Elimination gegenseitiger Rekursion

4. Elimination von Mustervergleichen

Der Mustervergleich (*pattern matching*) ist ein Konzept höherer Programmiersprachen und wird in einer maschinennahen Zielsprache wie Assembler oder C nicht unterstützt. Mustervergleiche werden vom $\mathcal{HDC}$–Compiler eliminiert, indem sie durch Abfragen ersetzt werden. Diese Vorgehensweise vereinfacht auch die nachfolgenden Phasen des Compilers.

Die für Mustervergleiche verwendeten `case`–Ausdrücke sind im Zwischencode bereits stark vereinfacht. Sie haben die Form:

$$
\begin{array}{l}
\texttt{case}\ Expr_0\ \texttt{of} \\
\quad pat_1\ \texttt{->}\ Expr_1 \\
\quad\ \vdots \\
\quad pat_n\ \texttt{->}\ Expr_n
\end{array}
$$

Ein Muster kann dabei nur noch ein einzelner Konstruktor mit der entsprechenden Anzahl von Argumenten sein. Für Listen und Tupel existieren spezielle vordefinierte Konstruktoren. Insbesondere sind Muster nicht mehr verschachtelt. Das letzte Muster pat_n kann auch aus einer Variablen bestehen.

In $\mathcal{HDC}$ werden `case`–Ausdrücke in verschachtelte `if-then-else`–Anweisungen umgewandelt. Die derzeitige Zielsprache C unterstützt zwar auch ein `case`–Konstrukt, mit dem mehrere `if`–Abfragen zusammengefaßt werden könnten. Jedoch ermöglicht die Verwendung von `if-then-else`, für das im abstrakten Code Sprünge eingefügt werden, eine Codegenerierung auch für andere Zielsprachen (wie z.B. Maschinensprache). Zudem vereinfachen einzelne Abfragen die Analyse für weitere Optimierungen.

Bei der Umwandlung eines `case`–Zweigs in den Zweig einer `if`–Abfrage müssen die Variablen des Musters substituiert werden. Es werden Zugriffe auf die Komponenten der Datenstruktur benötigt, mit der das Muster verglichen wird. Für Tupel und Konstruktoren algebraischer Datentypen kann dies auf die gleiche Weise mit einem speziellen Ausdruck geschehen. Die Variablen im Muster $(x{:}xs)$ für die Listenkonstruktion können durch entsprechende Aufrufe der Funktionen `head` und `tail` substituiert werden.

Im folgenden wird für die Substitution einer freien Variable v in einem Ausdruck E durch einen Ausdruck X die Schreibweise

$$E\texttt{[}\ v\ \texttt{:=}\ X\ \texttt{]}$$

verwendet.

4.1. Tupel und Konstruktoren

Der Zugriff auf die Komponenten eines Konstruktors bzw. Tupels erfolgt in der Implementierung von $\mathcal{HDC}$ mit einem besonderen Ausdruck `ENth`:

$$\texttt{(ENth } i\texttt{) (}C\ x_1\ \texttt{...}\ x_n\texttt{)} = x_i \qquad 1 \leq i \leq n$$
$$\texttt{(ENth } i\texttt{) (}x_1\texttt{, ..., } x_n\texttt{)} = x_i \qquad 1 \leq i \leq n$$

Für die Auswahl eines Zweiges wird der eindeutige Konstruktorindex verwendet, der in den Mustern gespeichert ist. Er wird mit dem Index des Prüfausdrucks verglichen der ebenfalls durch `ENth` ermittelt werden kann.

$$\texttt{(ENth 0) } \mathit{Expr} = \text{Index des Konstruktors in } \mathit{Expr}$$

`ENth` wird nur intern benötigt und kann aus Typisierungsgründen nicht in $\mathcal{HDC}$–Programmen verwendet werden. Es ist (abgesehen von Listenkonstruktoren) für die Konstruktoren aller Datentypen definiert.

Sei $I(i)$ der Konstruktorindex des Konstruktors C_i. Ein allgemeiner `case`–Ausdruck

$$
\begin{array}{l}
\texttt{case } \mathit{Expr}_0 \texttt{ of} \\
\quad C_1\ v_{(1,1)}\ \texttt{...}\ v_{(1,m(1))} \ \texttt{->}\ \mathit{Expr}_1 \\
\quad C_2\ v_{(2,1)}\ \texttt{...}\ v_{(2,m(2))} \ \texttt{->}\ \mathit{Expr}_2 \\
\quad \vdots \\
\quad C_n\ v_{(n,1)}\ \texttt{...}\ v_{(n,m(n))} \ \texttt{->}\ \mathit{Expr}_n \\
\quad \mathit{var} \qquad\qquad\qquad\quad\ \texttt{->}\ \mathit{Expr}_{n+1}
\end{array}
$$

läßt sich damit folgendermaßen in verschachtelte `if`–Abfragen umwandeln:

$$
\begin{array}{l}
\texttt{if (ENth 0) } \mathit{Expr}_0 \texttt{ == } I(1) \\
\texttt{then } \mathit{Expr}_1 \texttt{[} v_{(1,1)} \texttt{ := ((ENth 1) } \mathit{Expr}_0\texttt{), ...,} \\
\qquad\qquad\ \ v_{(1,m(1))} \texttt{ := ((ENth } m(1)\texttt{) } \mathit{Expr}_0\texttt{)]} \\
\texttt{else if (ENth 0) } \mathit{Expr}_0 \texttt{ == } I(2) \\
\qquad \texttt{then } \mathit{Expr}_2 \texttt{[} v_{(2,1)} \texttt{ := ((ENth 1) } \mathit{Expr}_0\texttt{), ...]} \\
\qquad \texttt{else ...} \\
\qquad\qquad \ddots \\
\qquad\qquad\qquad \texttt{else } \mathit{Expr}_{n+1} \texttt{[} \mathit{var} \texttt{ := } \mathit{Expr}_0 \texttt{]}
\end{array}
$$

Da es keine verschachtelten Muster mehr gibt, haben `case`–Ausdrücke für Tupel typischerweise nur einen Zweig: Das Muster ist ein Tupel mit gleichen Anzahl von Komponenten wie im Prüfausdruck. Einen weiteren Zweig für ein Tupel mit einer anderen Anzahl von Komponenten kann es in einem typkorrekten Programm nicht geben. Ein Zweig mit Variable ist überflüssig, da der Mustervergleich immer schon im ersten Zweig erfolgreich ist.

Anmerkung: Falls der letzte `case`–Zweig keine Variable enthält, wird das Muster des letzten Konstruktors ohne weitere Abfrage im letzten `else`–Zweig behandelt. Daraus kann sich ein Problem ergeben (siehe Abschnitt 4.3.2).

Es ist nicht optimal, die Abfragen für die Zweige in einer linearen Kette anzuordnen, wobei die Zweige des `case`–Ausdrucks von oben nach unten getestet werden. Besser wäre die Anordnung in einem Baum, indem das Verfahren der binären Suche angewendet wird. Dieser Aufwand lohnt sich jedoch erst, wenn die `case`–Ausdrücke mehrere Zweige haben. Statistische Auswertungen einiger Beispielprogramme zeigen jedoch, daß es pro `case`–Ausdruck im Schnitt weniger als zwei Zweige gibt. In Tabelle 4.1 sind die Anzahl der benötigten Funktionen, die Anzahl der darin enthaltenen `case`–Ausdrücke und die mittlere Anzahl von Zweigen pro `case`–Ausdruck aufgelistet. Es wurde der Zwischencode nach der Elimination funktionaler Argumente betrachtet. Aufgrund dieser Daten wurde die binäre Suche vorerst nicht implementiert.

	freqset_rek	freqset	freqhu	quicksort	pp_dc4io
Anzahl Funktionen	115	103	46	29	37
Zweige/`case`–Ausdr.	1,42	1,38	1,17	1,17	1,17
Anz. `case`–Ausdrücke	98	34	24	12	24
davon					
mit 1 Zweig	64	30	20	11	20
mit 2 Zweigen	32	2	4	0	4
mit 3 Zweigen	0	0	0	1	0
mit 4 Zweigen	0	0	0	0	0
mit 5 Zweigen	1	1	0	0	0
mit 6 Zweigen	1	0	0	0	0
mit 7 Zweigen	0	0	0	0	0
mit 8 Zweigen	0	1	0	0	0

Abbildung 4.1.: Mittlere Anzahl von Zweigen pro `case`–Ausdruck

4.2. Listen

Nach dem Löschen wiederholter Konstruktoren (vgl. Abschnitt 4.3.3) können in einem `case`–Ausdruck mit einer Liste als $Expr_0$ maximal drei Zweige übrig bleiben. Falls es drei Zweige gibt, hat der Benutzer neben den Zweigen für die leere Liste und Listenkonstruktion noch eine Variable angegeben. Eine Variable kann beim Auftreten in einem dritten Zweig ignoriert werden, weil der Mustervergleich immer bereits im ersten oder zweiten Zweig erfolgreich ist.

Normalerweise umfaßt ein Mustervergleich auf Listen zwei Zweige und kann folgendermaßen in eine `if`–Abfrage umgewandelt werden:

```
case Expr₀ of                  if null Expr₀
  []     -> Expr₁      ⇝         then Expr₁
  (x:xs) -> Expr₂                else Expr₂[ x  := (head Expr₀),
                                             xs := (tail Expr₀) ]
```

Die Variablen x und xs werden durch die Werte des ersten Elements und der Restliste von $Expr_0$ substituiert.

Wenn nur für einen Zweig ein Konstruktor angegeben ist, so muß die Variable im anderen Zweig entsprechend substituiert werden:

```
case Expr₀ of                  if null Expr₀
  []     -> Expr₁      ⇝         then Expr₁
  var    -> Expr₂                else Expr₂[ var := Expr₀ ]
```

bzw.

```
case Expr₀ of                  if null Expr₀
  (x:xs) -> Expr₁      ⇝         then Expr₂[ var := Expr₀ ]
  var    -> Expr₂                else Expr₁[ x  := (head Expr₀),
                                             xs := (tail Expr₀) ]
```

Anmerkung: Wie hier gezeigt, kann die Abfrage immer mit `null` erfolgen. Es ist nicht nötig, die Bedingung zu negieren. Gegebenenfalls müssen die beiden Zweige vertauscht werden. Das Vertauschen ist hier möglich, da die Zweige immer disjunkt sind.

4.3. Sonderfälle

4.3.1. Es existiert nur einen Zweig

Falls ein `case`–Ausdruck nur einen Zweig enthält, wird auf die Abfrage des Konstruktorindexes verzichtet, da es sowieso keine Alternative gibt. Es wird die rechte Seite des Zweigs verwendet, wobei im Muster auftretende Variablen substituiert werden. Die Korrektheit eines Programms wird dadurch erhalten. Ist der Mustervergleich aufgrund eines Programmierfehlers nicht erfüllbar, so ist das Ergebnis des Programm nicht vorhersagbar (vgl. Abschnitt 4.3.2).

Beispiel:

```
case Expr₀ of        ⇝     Expr₁[ x  := (head Expr₀),
  (x:xs) -> Expr₁                 xs := (tail Expr₀) ]
```

Hier entstehen keine Kosten für einen Vergleich, wenn eine Liste in (x:xs) aufgeteilt werden soll, die sicher nicht leer ist.

4.3.2. Es existiert kein passender Zweig

Es besteht die Gefahr, daß in einem vom Benutzer geschriebenen `case`–Ausdruck kein passendes Muster existiert. In diesem Fall wird das Programm abstürzen oder ein falsches Ergebnis liefern.

Wenn dieses undefinierte Verhalten nicht erwünscht ist, könnte das Programm durch zusätzliche Funktionen für Fehlerfälle angereichert werden, die dann eine entsprechende Fehlermeldung ausgeben. Diese Möglichkeit ist aber unabhängig von der Codeoptimierung und sollte gleich nach dem Parsen vorgenommen werden.

4.3.3. Konstruktoren werden wiederholt

Wenn in den Mustern der gleiche Konstruktor mehrfach auftritt, wird durch den Mustervergleich immer der Zweig des ersten Auftretens gewählt. Es können also alle Zweige mit wiederholten Konstruktoren gelöscht werden. Ein solcher Fall ist möglicherweise unbeabsichtigt oder durch einen Programmierfehler entstanden. Wenn die `case`–Elimination diesen Fall entdeckt, wird eine Warnung auf dem Bildschirm ausgegeben.

Beispiel:

```
case zs of                case zs of
  (x:xs) -> 0      ⇝       (x:xs) -> 0      ⇝      0
  (y:ys) -> y
```

5. Zwischencode für Wiederverwendung gemeinsamer Teilausdrücke

Der Zwischencode liegt in $\mathcal{HDC}$ nach den Transformationen zur Elimination von gegenseitiger Rekursion und `case`–Ausdrücken immer noch in Baumform vor. In diesem Baum treten gemeinsame Teilausdrücke mehrfach auf und werden daher mehrfach berechnet. In diesem Kapitel wird eine neue Zwischencode–Ebene eingeführt, in welcher gemeinsame Teilausdrücke nur einmal innerhalb des Rumpfs einer Funktion auftreten.

5.1. Funktionen als DAG

Die Umwandlung des Syntaxbaums einer Funktion in einen gerichteten azyklischen Graphen (DAG) ermöglicht die mehrfache Verwendung gemeinsamer Teilausdrücke. Der DAG einer Funktion besteht in der Implementierung von $\mathcal{HDC}$ aus einer nach Abhängigkeiten geordneten Menge von Ausdrücken. Jeder Ausdruck hat eine eindeutige Nummer innerhalb des DAGs. Bei zusammengesetzten Ausdrücken werden die Teilausdrücke durch die entsprechenden Nummern referenziert.

Beispiel:
Die folgende Funktion nimmt die ersten zwei Elemente einer Liste und addiert ihre Quadrate:

```
sqrplus1 xs = (xs!!0)*(xs!!0) + (xs!!1)*(xs!!1)
```

Für dieses Beispiel wurde die Funktion absichtlich umständlich kodiert, um mehr und größere gemeinsame Teilausdrücke zu erhalten. Andererseits benötigt der DAG für die vermeintlich bessere Definition

```
sqrplus2 xs = (xs!!0)^2 + (xs!!1)^2
```

sogar einen Knoten mehr für die Konstante `2`. Hierbei sind nur `xs` und `2` gemeinsame Teilausdrücke.

Abb. 5.1 zeigt den Datenfluß bei der Berechnung der Funktion `sqrplus1`. Im Datenflußgraphen lassen sich gemeinsame Teilausdrücke durch mehrere Kanten ausgehend von einem Knoten darstellen.

In der Abbildung ist ebenfalls zu sehen, wie der DAG im Zwischencode als Tabelle abgelegt ist. Die Zeilen sind durchnummeriert und enthalten Ausdrücke. Hier ist die Wurzel des DAGs der Ausdruck mit Nummer 7, der formale Parameter `xs` ist in Zeile 0 zu finden. Die Referenzen auf Teilausdrücke werden im folgenden mit einem Pfeil (↑) gekennzeichnet. Sie entsprechen den Kanten des Datenflußgraphen in umgekehrter Richtung.

```
sqrplus1 :: [Int] -> Int
sqrplus1 xs
ROOT = 7 FORMPAR = [0]
0:  xs              :: [Int]
1:  0               :: Int
2:  (!!)   ↑0 ↑1    :: Int
3:  (*)    ↑2 ↑2    :: Int
4:  1               :: Int
5:  (!!)   ↑0 ↑4    :: Int
6:  (*)    ↑5 ↑5    :: Int
7:  (+)    ↑3 ↑6    :: Int
```

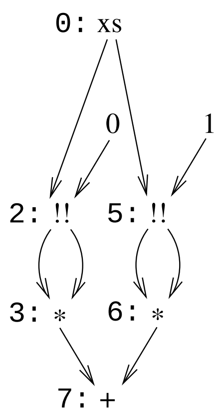

Abbildung 5.1.: Beispiel für den Aufbau eines DAGs

5.2. Konstruktion der DAGs

Zur Konstruktion der Knoten für den DAG einer Funktion wird die Zahlenwert-Methode (*value-number method*) [ASU86, Seite 292] verwendet:

Der Algorithmus benötigt eine Datenstruktur, in der auf Knoten über Nummern zugegriffen wird. Aus Effizienzgründen wurde hierfür ein AVL-Baum [AHU74] gewählt. Ein AVL-Baum ist ein balancierter Baum mit Zugriffszeit auf ein Element in $O(\log n)$, n ist die Anzahl der Elemente im Baum. Die Arrays von Haskell sind hier nicht flexibel genug, weil beim Anlegen die Größe angegeben werden muß, die danach nicht mehr verändert werden kann.

Algorithmus (Einfügen eines Knotens in den DAG)

Eingabe: Ausdruck *expr*, Teilausdrücke schon durch Nummern referenziert
Ausgabe: Nummer für *expr*
Methode: Suche *expr* im AVL-Baum. Wurde ein Knoten gefunden, der mit *expr* übereinstimmt, dann gib dessen Nummer zurück. Andernfalls, erzeuge neuen Knoten und liefere dessen Nummer.

```
lookupDAG :: DAG -> Expr -> Maybe Int
lookupDAG (DAG _ d) e
  = maybe Nothing (Just . fst) (avl_findfirst cond d)
    where cond (_,e') = cmpExpr e' e

insertDAG :: DAG -> Expr -> (DAG,Int)
insertDAG (DAG count d) e
  = (DAG (count+1) (avl_insert (count,e) d),count)

makeNode :: DAG -> Expr -> (DAG,Pointer)
makeNode dag expr
  = case lookupDAG dag expr of
      Just i  -> (dag,i)
      Nothing -> insertDAG dag expr
```

Abbildung 5.2.: Funktion zur Erzeugung eines DAG Knotens

Mit einer rekursiven Funktion, die zuerst Teilausdrücke umwandelt und dann den Knoten mit `makeNode` (siehe Abb. 5.2) erzeugt, kann ein Ausdruck von der Syntaxbaumdarstellung in den DAG–Zwischencode umgewandelt werden. Die Definition von `cmpExpr` für den syntaktischen Vergleich zweier Ausdrücke wird hier nicht wiedergegeben, da sie offensichtlich ist: Konstanten sind gleich, wenn sie gleiche Werte enthalten. Zusammengesetzte Ausdrücke sind gleich, wenn die Ausdrücke selbst und alle Zeiger auf Teilausdrücke übereinstimmen. Zusätzlich werden die Typen auf Gleichheit geprüft. Dies ist für den Konstruktor `[]` und `ENth`–Ausdrücke unbedingt erforderlich, weil sich diese im Typ unterscheiden können, obwohl alle anderen Felder gleich sind.

Die erzeugten DAGs erfüllen folgende wichtige Eigenschaften:

- Gleiche Ausdrücke treten nicht mehrfach auf.
- Ein Ausdruck mit der Nummer i in einem DAG hängt höchstens von Ausdrücken mit Nummern $j < i$ ab, d.h. die Ausdrücke im DAG sind nach Abhängigkeiten sortiert.

Die Umwandlung wird für die Definitionen aller benötigten Funktionen durchgeführt. Während der Umwandlung wird in einer Liste gespeichert, welche Nummern die Ausdrücke der formalen Parameter im DAG erhalten und welche Nummer der Wurzelausdruck erhält. In der Liste wird der entsprechende Eintrag eines formalen Parameters mit der internen Konstante `undef` markiert, falls dieser nicht im Funktionsrumpf verwendet wird.

Die gespeicherten Informationen werden in der Funktionsdefinition abgelegt. Beim Expansionsalgorithmus wird ein Zugriff auf die Informationen benötigt, dazu werden die folgenden Funktionen verwendet:

`size` f	liefert die Anzahl der Ausdrücke im DAG von f
`root` f	liefert Referenz auf die Wurzel des DAGs von f
`args` f	liefert Referenzen auf formale Parameter von f
`list` f	geordnete Liste aller (Nummer, Ausdruck)–Paare im DAG

5.3. Zwischencode–Transformationen

Bei einer Transformationen auf einem DAG ist es wichtig, daß die Abhängigkeiten im DAG erhalten bleiben: Ein Ausdruck an Position i darf nur Ausdrücke mit Positionen $0 \ldots i-1$ als Teilausdrücke verwenden.

Diese Bedingung kann beim Umformen eines DAGs sichergestellt werden, indem eine Kopie angelegt wird: Von einem Quell-DAG Δ_q werden nacheinander die Ausdrücke gelesen und in einen (anfangs leeren) Ziel-DAG Δ_z eingefügt. So bleiben die Abhängigkeiten erhalten, da sich die Reihenfolge der Ausdrücke relativ zueinander nicht ändert. Es können dabei auch neue Ausdrücke, die nicht in Δ_q existieren, in Δ_z angelegt werden. Umgekehrt ist es möglich, Ausdrücke selektiv einzufügen, was dem Löschen von Ausdrücken entspricht. Im letzteren Fall muß jedoch beachtet werden, daß keine Ausdrücke gelöscht werden, die noch benötigt werden.

Durch Einfügen oder Löschen verschieben sich Ausdrücke im DAG. Sie erhalten dadurch eine andere Nummer und es müssen Referenzen auf veränderte Nummern angepaßt werden. Die Anpassung erfolgt mit Hilfe einer Tabelle τ. Ihre Größe entspricht der Größe von Δ_q. In der Tabelle wird für die Nummer jedes Ausdrucks aus Δ_q gespeichert, welche Nummer er durch das Einfügen in Δ_z erhält.

Das Kopieren eines DAGs wird mit der Funktion `insert` aus Abb. 5.3 durchgeführt: Sie fügt einen Knoten aus dem Quell-DAG in den Ziel-DAG ein. Dabei wird die übergebene Tabelle τ aktualisiert. Durch wiederholtes Einfügen wird die Tabelle stückchenweise gefüllt.

Beim Einfügen des Ausdrucks in den Ziel-DAG ist zu beachten, daß die Referenzen auf Teilausdrücke angepaßt werden müssen. Diese Abbildung wird von der Funktion Φ geleistet. Die Funktion benötigt die Tabelle τ und den alten Wert einer Referenz (Nummer aus Quell-DAG). Sie liefert die Nummer des referenzierten Ausdrucks im Ziel-DAG. Die Zeigeranpassung mit Hilfe einer teilweise ausgefüllten Tabelle kann immer erfolgen, weil alle Teilausdrücke – aufgrund der Ordnung nach Abhängigkeiten im DAG – bereits kopiert wurden. Für die Teilausdrücke ist daher der entsprechende Eintrag in der Tabelle schon vorhanden.

In der Funktionsdefinition, in der der Ziel-DAG schließlich verwendet wird, müssen auch die Nummern der formalen Parameter angepaßt werden. Dies erfolgt ebenfalls mit Hilfe der Tabelle. Argumente die nicht im Funktionsrumpf verwendet werden, bleiben unverändert.

Im folgenden wird die Kurzschreibweise Φ_τ anstelle von (Φ τ) für die Funktion Φ mit τ als erstem Parameter verwendet. Hiermit soll angedeutet werden, daß die Abbildung im Vordergrund steht, die als Hilfsmittel die Tabelle τ verwendet.

Für Transformationen zur Vereinfachung von Ausdrücken ist es oft hilfreich, Identitätsknoten zu verwenden. Sie verweisen auf einen anderen Ausdruck im DAG, mit dem sie identisch sein sollen. Solche Knoten verletzen temporär die Eigenschaft, daß gleiche Ausdrücke nicht an mehreren Stellen im DAG gespeichert sind. Wird versucht, mit der Funktion `insert` einen Identitätsknoten einzufügen, so wird dieser eliminiert: der DAG wird unverändert zurückgegeben und die Tabelle wird mit der Nummer des Original–Knotens im Ziel–DAG aktualisiert. Referenzen auf den Identitätsknoten in den nachfolgenden Ausdrücken werden dann beim Einfügen (durch Anpassung mit der Funktion Φ) in Referenzen auf den Original–Knoten umgesetzt. Auf diese Weise werden durch Vereinfachungen entstandene Identitätsknoten bei der nächsten Transformation, die den DAG mit Hilfe von `insert` kopiert, entfernt. Dies erfolgt in der Optimierungschleife spätestens bei der Transformation zur Entfernung unerreichbaren Codes (siehe Kapitel 6). Es wird dadurch sichergestellt, daß im abstrakten Code keine Identitätsknoten mehr auftreten.

```
type Table = Array Int Int

Φ :: Table -> Int -> Int
Φ τ pos
  = if pos == undef then undef else τ ! pos

insert :: (DAG,Table) -> (Int,Expr) -> (DAG,Table)
insert (Δ,τ) (pos,e)
  = if e ist Identitätsknoten
    then (Δ, τ // [(pos,Φτ(Nummer des Original–Knotens)])
    else let
      e′ = e mit Φτ angewendet auf Nummern der Teilausdrücke
      (Δ′, pos′) = makeNode Δ e′
      τ′ = τ // [(pos,pos′)]
    in (Δ′, τ′)
```

Abbildung 5.3.: Definition der Funktionen Φ und `insert`

6. Optimierungsschleife

Die Codeoptimierung erfolgt in einer Schleife. In jeder Iteration werden Funktionsaufrufe expandiert, Vereinfachungsregeln angewendet und Größeninferenz durchgeführt. Abb. 6.1 zeigt in der Übersicht wie diese unabhängigen Phasen durchlaufen werden.

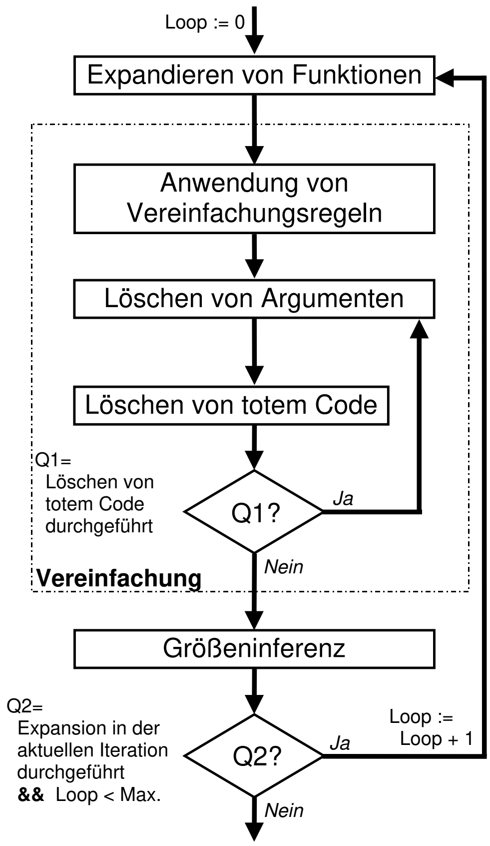

Abbildung 6.1.: Verschaltung der Phasen innerhalb der Optimierungsschleife

Die Schleife wird verlassen, wenn im unmittelbar vorausgegangenen Schleifendurchlauf keine Funktion mehr expandiert werden konnte oder eine maximale, benutzerdefinierbare Anzahl an Wiederholungen erreicht wurde.

6.1. Expansion von Funktionen

6.1.1. Vorbemerkungen

Die Expansion eines Funktionsaufrufs bezeichnet den Vorgang, bei dem der Funktionrumpf an die Aufrufstelle kopiert wird und dabei die formalen durch die aktuellen Parameter substituiert werden. Mit Expansion einer Funktion ist gemeint, daß Funktionsaufrufe in dieser Funktion expandiert werden.

Die referentielle Transparenz von Haskell garantiert, daß diese Transformation die Korrektheit des Programms erhält. Weiterhin besteht keine Gefahr, daß sich der Rechenaufwand vergrößert, wenn auch bei der Expansion gemeinsame Teilausdrücke nur einmal im DAG abgelegt und nicht dupliziert werden.

Durch die Expansion werden Funktionsaufrufe eliminiert und der damit verbundene Overhead reduziert. Funktionen können gelöscht werden, wenn sie vom Programm nicht mehr aufgerufen werden.

Wichtiger ist jedoch, daß durch Expansion die Lokalität erhöht wird und dadurch weitere Ausdrücke gemeinsam genutzt werden können. Dies ist im $\mathcal{HDC}$–Compiler deswegen von großer Bedeutung, weil damit die `let`–Ausdrücke effizienter implementiert werden können.

Standardmäßig werden alle gegenseitigen Rekursionen eliminiert. Der Benutzer kann diese Einstellung jedoch verändern. Ein wichtiger Aspekt für das Design des Expansionsalgorithmus ist deshalb, daß auch für Programme, die gegenseitige Rekursionen enthalten, die Termination des Verfahrens gewährleistet sein muß.

6.1.2. Vorgehensweise bei der Expansion

Es wurde die folgende Vorgehensweise gewählt: Zur Expansion einer Funktion f wird ihr DAG von oben nach unten durchlaufen, und die Ausdrücke werden mit Hilfe der in Kapitel 5.3 vorgestellten Funktionen in einen neuen DAG übertragen. Wird ein Ausdruck erreicht, der einen Funktionsaufruf einer Funktion g enthält, so wird eine Entscheidung darüber gefällt, ob dieser expandiert werden darf, oder nicht. Wird er expandiert, so werden in den neuen DAG von f die Ausdrücke des DAGs von g eingefügt. Hierbei werden *nicht* weitere Funktionsaufrufe rekursiv expandiert, wodurch die Termination gesichert ist. Funktionsaufrufe, die im DAG von g enthalten sind, werden also grundsätzlich kopiert. Sie können jedoch in einem der folgenden Schleifendurchgänge expandiert werden.

Abb. 6.2 zeigt ein Beispiel für die beschriebene Vorgehensweise. Hier sind DAGs durch eine Tabellenform dargestellt. Der DAG für die Funktion f enthält in Zeile 2 einen Ausdruck für den Funktionsaufruf der Funktion g. In f sollen Funktionsaufrufe expandiert werden. Dazu wird die Tabelle der Ausdrücke für f von oben nach unten durchlaufen. Es werden dabei die Ausdrücke in eine neue Tabelle kopiert, die später die aktuelle Tabelle des DAGs von f ersetzt. Ausdruck 0 und 1 werden kopiert und erhalten in der neuen Tabelle ebenfalls die Positionen 0 bzw. 1. Nun wird ein Funktionsaufruf angetroffen. Um diesen zu expandieren, werden im folgenden die Ausdrücke der Tabelle für g in die neue Tabelle für f kopiert.

Es ist dabei möglich, daß ein Ausdruck gefunden wird, der schon in der neuen Tabelle vorhanden ist (gestrichelter Pfeil). In diesem Fall wird keine Kopie des Ausdrucks angelegt. Es wird nur für nachfolgende Refernzierungen dieses Ausdrucks festgehalten, in welcher Zeile des neuen DAGs er nun zu finden ist.

Nachdem alle Ausdrücke aus g bearbeitet wurden, werden die weiteren Ausdrücke von f durchlaufen und kopiert. Es können nun auch gemeinsame Teilausdrücke mit dem aus g kopierten Code auftreten. Ein solcher Fall ist wiederum mit einem gestrichelten Pfeil angedeutet.

In der Tabelle von f können auch weitere Funktionsaufrufe auftreten. Falls eine entsprechende Bedingung erfüllt ist, können diese auf die gleiche Weise expandiert werden.

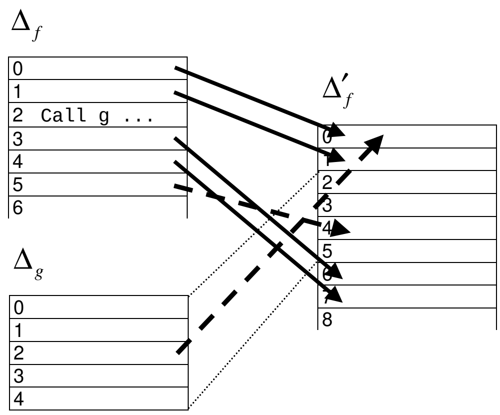

Abbildung 6.2.: Vorgehensweise bei der Expansion

In Abb. 6.3 ist der Algorithmus angegeben, mit dem die Funktionsaufrufe expandiert werden. `dagCopyExpand` werden alle Ausdrücke der Funktion, die expandiert werden soll, nacheinander übergeben. `dagCopyExpand` bekommt zudem den Ziel-DAG Δ_f und die Tabelle τ. Beide sind zu Anfang leer und werden schrittweise gefüllt. In Zl. 0 wird geprüft, ob es sich um einen expandierbaren Funktionsaufruf handelt.

Wenn der aktuelle Ausdruck kein Funktionsaufruf ist, oder die Entscheidung, den Funktionsaufruf zu expandieren, negativ ausfällt, so wird in den neuen DAG wieder dieser Ausdruck eingefügt. Das Einfügen (Abb. 6.3, Zl. 9) erfolgt mit der Funktion `insert` aus Abb. 5.3. Sie führt die nötigen Zeigeranpassungen durch und fügt gemeinsame Teilausdrücke nicht mehrfach ein.

Wenn der Fall eintritt, daß ein Funktionsaufruf expandiert wird, so müssen die Zeiger auf die Werte der Parameter im Funktionsaufruf an die veränderten Positionen im neuen DAG angepaßt werden. Diese Werte sind bereits (auf Grund der Abhängigkeiten) in den Ziel-DAG kopiert worden und können durch Φ_τ angepaßt werden (Zl. 3).

Es wird für den zu kopierenden DAG von `g` auch eine Tabelle (hier: υ) benötigt, mit der Zeigeranpassungen durch die Funktion Φ erfolgen können. In Zl. 6 wird eine leere Tabelle in der Größe des DAGs von `g` angelegt und für Positionen der formalen Parameter von `g` eingetragen, wo sich die aktuellen Werte im neuen DAG befinden (es sind die angepaßten Referenzen aus Zl. 3).

Nun können die Ausdrücke von `g` mit `insert` kopiert werden (Zl. 7). Dabei

```
   emptyTable count = listArray (0,count-1) (repeat undef)

   dagCopyExpand :: ((DAG,Table),Bool) -> (Int,Expr)
                    -> ((DAG,Table),Bool)
   dagCopyExpand ((Δ_f,τ),again) (i_f,e_f)
     = if e_f ist Funktionsaufruf g ↑x_1 ... ↑x_n ∧ g ist kein Skelett
0          ∧ expandAllowed
1      then
2        let
3          actpars = map Φ_τ [ ↑x_1,...,↑x_n ]
4          subst = filter ((/=undef).fst) (zip (args g) actpars)
5          body_exprs = filter ((/=EName).ec.snd) (list g)
6          υ = emptyTable (size g) // subst
7          (Δ'_f,υ') = foldl insert (Δ_f,υ) body_exprs
8        in ((Δ'_f,τ//[(i,Φ_υ'(root g))]), True)
9      else (insert (Δ_f,τ) (i_f,e_f), again)
```

Abbildung 6.3.: Algorithmus zur Expansion von Funktionsaufrufen müssen jedoch die Ausdrücke, die formale Parameter enthalten, herausgefiltert werden (Zl. 5), da deren Werte bereits im DAG und der Tabelle v eingetragen sind. Wird beim Kopieren von `g` ein Ausdruck erreicht, der einen formalen Parameter als Teilausdruck referenzierte, so wird der Referenz durch den Aufruf von Φ von `insert` auf die Nummer gesetzt, wo der Wert des Übergabeparameters zu finden ist.

Ist das Kopieren der Ausdrücke von `g` beendet, so wurde als letzter Ausdruck die Wurzel von `g` kopiert. Ihre Nummer im neuen DAG wird in die Tabelle τ an die Stelle des früheren Aufrufs von `g` eingetragen (Zl. 8). Auf diese Weise werden nun alle folgenden Ausdrücke, die den Funktionsaufruf von `g` als Teilausdruck hatten, das direkt in Funktion `f` berechnete Ergebnis verwenden.

Die Entscheidung ob ein Aufruf expandiert wird, berücksichtigt zur Zeit nur, ob der maximal zu erwartende Codezuwachs im Rahmen der vom Benutzer festgelegten Grenzen liegt (vgl. Abb. 6.4). Eine obere Schranke für die Größe einer Funktion nach der Expansion eines Funktionsaufrufs ist gegeben durch ihre aktuelle Größe in DAG–Knoten, zuzüglich der Größe der zu kopierenden Funktion. Da der ursprüngliche Funktionsaufruf wegfällt, kann noch ein Knoten abgezogen werden.

Der Benutzer kann eine maximale relative und eine maximale absolute Größe festlegen. Die relative Maximalgröße gibt an, auf ein Wievielfaches der Originalgröße – unmittelbar nach der Umwandlung in den DAG–Zwischencode – jede Funktion anwachsen darf. Die absolute Maximalgröße gibt eine feste Grenze an, über die keine Funktion im DAG–Zwischencode hinaus wachsen darf.

```
expandAllowed =
  let myMaxNewSize = myCurSize + inlSize - 1
  in  (myMaxNewSize <= inlMaxRelFunSize * myOrigSize) &&
      (myMaxNewSize <= inlMaxAbsFunSize)
```

Abbildung 6.4.: Entscheidung für oder gegen Expansion

Das absolute Maximum für die Anzahl der DAG–Knoten einer Funktion ist dadurch motiviert, daß

- übermäßig große Funktionen ineffizient in der Ausführung sind, falls der Prozessor–Cache nicht ausreichend groß ist,
- die Codeoptimierung mehr Rechenzeit bei großen Funktionen benötigt,
- durch die Elimination gegenseitiger Rekursion bereits sehr große Funktionen entstehen können, eine ausschließlich relative Größenbeschränkung könnte dann zu einem unerwartet hohen Codezuwachs führen.

6.1.3. Die Strategien OVI und CVI

Die Auswahl der Funktionsdefinition, die bei der Expansion eines Funktionsaufrufs verwendet wird, kann in $\mathcal{HDC}$ wahlweise durch zwei unterschiedliche Methoden erfolgen: Entweder wird die Originalversion der Funktionsdefinition verwendet, die nach Umwandlung des Syntaxbaums in den DAG vorliegt. Oder es wird die gerade aktuelle Version verwendet, die durch Expansion und Vereinfachung der Originalversion entstanden ist. Kaser et al. [KRP92] bezeichnen diese Methoden als *original version inlining* (hier: **OVI**) bzw. *current version inlining* (hier: **CVI**) und weisen theoretisch und experimentell nach, daß durch die Verwendung der Originalversionen schnellere Programme erzeugt werden können. Sie zeigen auch, daß ohne Vorliegen von Rekursion die Verwendung der Originalversionen keine Vorteile bringt.

CVI hat den Vorteil, daß weniger Expansionsvorgänge benötigt werden, weil bereits expandierte Funktionen weiterverwendet werden. Die Optimierungsschleife kommt daher in der Regel schneller zum Ende. Allerdings kann die Codegröße bei der Expansion von rekursiven Funktionsaufrufen exponentiell wachsen, und dadurch können nicht alle Programme erzeugt werden, die durch Expansion mit **OVI** (bei linearem Codezuwachs) möglich sind.

Bei **OVI** ist zu beachten, daß in der Optimierungsschleife keine Funktionen gelöscht werden dürfen. Es ist denkbar, daß das Programm eine Funktion f zu einem Zeitpunkt nicht mehr benötigt. Angenommen eine Funktion g enthält in ihrer Originalversion einen Aufruf von f. Wird zu einem späteren Zeitpunkt ein Funktionsaufruf von g expandiert, so taucht wieder ein Funktionsaufruf von f in dem Programm auf. Weiterhin dürfen auch keine nicht benötigten Funktionsargumente in der Schleife gelöscht werden, da sonst die Typen der Originalversionen nicht mehr mit den aktuellen Versionen übereinstimmen und Expansion unmöglich wird. (Nach der Schleife ist das Löschen möglich.)

Im folgenden wird die Effizienz **CVI** und **OVI** verglichen. In Tabelle 6.1 sind die Optimierungsergebnisse für ausgewählte Beispielprogramme gegenübergestellt. Die Programme und Informationen über ihre Größe ohne Optimierung sind im Anhang aufgelistet.

Es wurden folgende Compilereinstellungen verwendet:

`Mutual Recursion Elim.:`	`NONE`
`Optimize By Inlining:`	`True`
`Algorithm:`	**OVI** bzw. **CVI**
`Number Of Loops:`	`4`
`Max. Rel. Funsize:`	`4 * original size`
`Max. Abs. Funsize:`	`200 nodes`
`Inline Order:`	`InlOrderIncSize`

Eingaben für den Interpreter:

freqset und freqset_rek	`0.3` *bag*
freqhu	`0.3 [1,2,3,4,5,6]` *bag*
quicksort	`[20,19..0]`

bag = `[[2,3],[2,3,4],[2,3,4,5],[2,3,4,5,6],[3,4],[3,4,5],[4,5,6]]`

	freqset		freqset_rek		freqhu		quicksort	
	OVI	**CVI**	**OVI**	**CVI**	**OVI**	**CVI**	**OVI**	**CVI**
Fkt	31	25	73	54	18	10	9	8
Kn	577	458	1770	1380	296	219	71	65
Op	33126	31703	45612	39465	3847	3333	4197	4196
Sch	566	552	4907	4632	422	366	466	465
Par	58.5	57.4	9.3	8.5	9.1	9.1	9.0	9.0

Tabelle 6.1.: Vergleich von Expansion mit Original– / aktueller Version

Mit diesem Beispiel kann nicht bestätigt werden, daß **OVI** für hochrekursive Programme besser geeignet ist. Es fällt auf, daß mit **OVI** optimierte Programme mehr Funktionen enthalten. Diese Eigenart ist auch bei anderen Beispielprogrammen zu beobachten und liegt vermutlich daran, daß bereits eliminierte Funktionsaufrufe bei der Expansion mit Originalversionen wieder in das Programm kommen oder erst gar nicht daraus eliminiert werden können.

Es ist hier ebenfalls zu beobachten, daß der Parallelitätsgrad in der rekursiven Version viel geringer ist. Dieser Effekt ist unabhängig davon, ob expandiert wird oder nicht. Er entsteht durch die größere Anzahl der Kontrollabhängigkeiten in der rekursiven Version und ist ein weiterer Grund, neben den Effizienzproblemen bei der Elimination gegenseitiger Rekursion, möglichst die in der Prelude vordefinierten Funktionen zu verwenden und eigene rekursive Definitionen zu vermeiden.

Bei **CVI** wird die Möglichkeit der Expansion eines Funktionsaufrufs von der Reihenfolge beeinflußt, in der die Funktionen expandiert werden. Soll in einer Funktion f_1 ein Funktionsaufruf von f_2 expandiert werden, so muß geprüft werden, ob dies erlaubt ist. Die Entscheidung hängt von der momentanen Größe von f_1 und der Größe von f_2 ab. Wurde die Funktion f_2 bereits vor f_1 bearbeitet (und ist sie dadurch größer geworden), so ist es möglich, daß keine Expansion durchgeführt werden darf. Andererseits könnte der Aufruf von f_2 unter Umständen expandiert werden, falls f_2 nach f_1 bearbeitet wird.

Wenn für die Expansion stets die unveränderten Originalversionen der Funktionen verwendet werden, spielt die Reihenfolge keine Rolle.

Bei den Experimenten im Anhang wird untersucht, welchen Einfluß die Reihenfolge der Expansionen auf den resultierenden optimierten Code von Beispielprogrammen hat.

Folgende Reihenfolgen werden betrachtet:

InlOrderTopSort
: Die topologische Sortierung ordnet die Funktionen so, daß eine Funktion immer vor den Funktionen steht, in denen sie aufgerufen wird. Diese Ordnung ist nur möglich, wenn keine gegenseitige Rekursion vorliegt.

InlOrderCallGraph
: Die umgekehrte Reihenfolge, in der die Funktionen beim Aufbau des Auf-

rufgraphen gefunden wurden. Sie ist der topologischen Sortierung ähnlich, weil Funktionen häufig vor der Funktion stehen, in der sie aufgerufen werden. Es gibt aber Sonderfälle in denen diese Eigenschaft nicht erfüllt ist.

InlOrderFunNumber
Die Funktionen werden nach Reihenfolge ihrer Definitionen sortiert.

InlOrderIncSize
Die Sortierung erfolgt nach Anzahl der DAG–Knoten der Funktionen. Kleine Funktionen stehen am Anfang, große am Ende dieser Reihenfolge.

InlOrderDecSize
Die Sortierung erfolgt in umgekehrter Reihenfolge wie bei InlOrderIncSize, also nach absteigender Größe.

InlOrderRandom
Es wird eine zufällige Reihenfolge der Funktionen erzeugt. Diese Auswahl kann sehr unterschiedliche Ergebnisse liefern. Das Programm kann mehrmals optimiert werden und ein Durchschnitt für die ermittelten Werte gebildet werden. Dieser ist als Vergleichswert interessant.

In Tabelle B.1 werden die verschiedenen Reihenfolgen bei **CVI** für die Programme `freqset.hdc` und `freqset_rek.hdc` verglichen. (InlOrderTopSort wurde nicht untersucht, da Elimination gegenseitiger Rekursion wegen Effizienzproblemen bei `freqset_rek` deaktiviert wurde. Siehe Kapitel 3.4) Eingabe und Compilereinstellungen waren gleich wie für Tabelle 6.1. Es fällt auf, daß der abstrakte Code des optimierten Programms am kleinsten bei der Reihenfolge InlOrderIncSize ist. (Verglichen mit Mittelwert der zufälligen Reihenfolgen: Einsparung von 14.4% bzw. 10%) Die Anzahl der ausgeführten Operationen wird bei `freqset` kaum von der Reihenfolge des Expansion beeinflußt. Bei `freqset_rek` sind die Schwankungen jedoch ziemlich stark. Im Vergleich ist auch hier die Reihenfolge InlOrderIncSize am günstigsten: Es werden vom Mittelwert der zufälligen Reihenfolgen noch 7% an Operationen eingespart.

Da, wie bereits erwähnt wurde, die Reihenfolge bei **OVI** keine Auswirkungen hat, ist für dieses Verfahren keine Vergleichstabelle nötig. Experimentelle Ergebnisse weiterer Beispielprogramme befinden sich im Anhang.

6.2. Vereinfachungen des Zwischencodes

Mit Expansion von Funktionen bietet eine Menge weiterer Optimierungsmöglichkeiten. Es entstehen zum Teil auch Kombinationen von Ausdrücken, die ungeschickt sind und offensichtlich vereinfacht werden können. Solche Ausdrücke werden normalerweise nicht in einem Benutzerprogramm auftreten. Ein Beispiel hierfür ist `[0..100]!!5`, was zu `5` vereinfacht werden kann.

Welchen Stellenwert das Vereinfachen in einem optimierenden Compiler für eine funktionale Sprache einnimmt, kann am Glasgow Haskell Compiler [JHH+93] beobachtet werden. Die Module für die Vereinfachung auf zwei Zwischencode–Ebenen (*Core language* und *STG language*) umfassen in der aktuellen Version GHC 4.02 ca. 11000 Zeilen Haskell–Sourcecode (mit Kommentaren). Das entspricht 12% des Codes des gesamten Compilers. Im Vergleich dazu, umfaßt der gesamte praktische Teil dieser Diplomarbeit ca. 2000 Zeilen Haskell–Code.

Die in den folgenden Abschnitten vorgestellten Vereinfachungen sind natürlich nur ein Bruchteil dessen, was möglich ist. Es können hier noch Erweiterungen vorgenommen werden, um die Effizienz des optimierten Codes weiter zu steigern.

Für die Vereinfachungen wird nach dem Vorkommen bestimmter Strukturen in den DAGs gesucht und diese werden durch äquivalente, einfachere Ausdrücke ersetzt. Die Ersetzungsregeln werden durch Abbildungen veranschaulicht. Dabei wird auf der linken Seite ein Ausschnitt des DAGs dargestellt, der durch den Graph auf der rechten Seite ersetzt wird. Die Vereinfachungen reduzieren die Anzahl der DAG–Knoten, die für die Berechnung von Ausdrücken benötigt werden. Dadurch kann unerreichbarer Code entstehen, falls die Knoten, auf die eine Referenz durch die Vereinfachung wegfällt, an keiner anderen Stelle im DAG benötigt werden. Der tote Code wird einmal in jedem Schleifendurchlauf der Optimierungschleife, nach Anwendung aller Vereinfachungsregeln entfernt. Nach jeder Regel wird der DAG von mehrfach auftretenden Ausdrücken bzw. Identitätsknoten gesäubert. Dies wäre nicht unbedingt erforderlich, erleichtert aber die weitere Suche nach vereinfachbaren Strukturen.

6.2.1. Elimination von Indexoperationen

Als regulär werden hier Listen bezeichnet, die durch folgende Berechnungsvorschriften in $\mathcal{HDC}$ erzeugt werden können:
`enumFromTo a b` wird intern für die arithmetische Sequenz `[a..b]` verwendet.
`sinGen f n` ist definiert als `map f [0..n-1]`. [HLG+99]

Für die regulären Listen ist die Struktur bekannt, und die Indizierung eines Elements kann in eine direkte Berechnung umgewandelt werden. Diese Umwandlung ist insbesondere bei einer strikten Sprache nützlich, weil in diesem Fall die gesamte Liste berechnet werden muß, bevor der Indexoperator darauf angewendet werden kann. Damit werden unter Umständen viele unnötige Berechnungen durchgeführt.

Die Abb. 6.5 zeigt, wie eine Indizierung dieser regulären Listen vereinfacht wird. Zu beachten ist hierbei, daß an `sinGen` ein Funktionscode anstelle einer Funktion übergeben wird. Zur Auswertung der kodierten Funktion wird eine apply–Funktion [Sch98] benötigt. Sie wurde vor der Vereinfachung von dem Skelett aufgerufen, durch das `sinGen` implementiert ist. Welche Funktion benötigt wird, läßt sich aus dem Funktionscode ermitteln.

Für diese Vereinfachung ist unerheblich, ob `a`, `b`, `n` und `x` als Konstanten vorliegen oder durch zusammengesetzte Ausdrücke berechnet werden.

Bei arithmetischen Sequenzen wird, falls `a` die Konstante `0` enthält, der Ausdruck des Indexoperators durch einen Identitätsknoten für `x` ersetzt. Enthält `a` eine andere Konstante oder ist unbekannt, so muß der Anfangswert der Sequenz zur Position addiert werden.

Die beiden Vereinfachungen erhalten die Korrektheit des Programms. Problematisch kann es jedoch werden, wenn der Index außerhalb der Liste liegt. Angenommen die Funktion `(!!)` enthält dafür eine Fehlerabfrage: Durch die Vereinfachung würde die Fehlerausgabe der Optimierung zum Opfer fallen. Um die Sicherheit weiterhin zu gewährleisten, müßte daher, falls die Einhaltung der Grenzen nicht statisch geprüft werden kann, ein Vergleich des Indexes mit der Länge der Liste erzeugt werden. In der Implementierung von $\mathcal{HDC}$ werden derzeit keine Fehlerabfragen gemacht, damit das Programm möglichst schnell ablaufen kann.

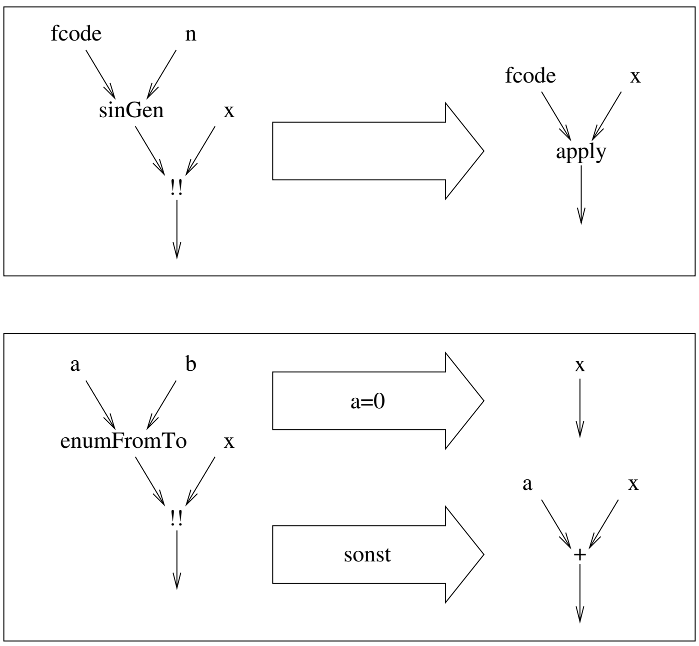

Abbildung 6.5.: Vereinfachung von Indizierung regulärer Listen

6.2.2. Elimination von Konstruktoren

Eine ähnliche Vereinfachung, wie für reguläre Listen gezeigt, kann auf `ENth`–Ausdrücke, die durch die Elimination der Mustervergleiche entstanden sind (siehe Kap. 4), angewendet werden. Diese Ausdrücke dienen zur Selektion eines Arguments eines Konstruktors.

Der `ENth`–Ausdruck kann durch den Wert des Arguments ersetzt werden, wenn der Konstruktor bekannt ist. In diesem Fall ist das erste Argument von `ENth` eine Konstruktorapplikation und nicht eine Variable, die einen Konstruktor enthält. Eine derartige Konstellation ergibt sich in der Regel erst, wenn bereits Funktionen expandiert wurden.

Diese Optimierung ist unkritisch, da die `ENth`–Ausdrücke nur intern verwendet werden und die Auswahl des Elements immer in einem gültigen Bereich liegt.

Die Auswertung von `(ENth 0)` liefert den Konstruktorindex. Es wird ein Ausdruck mit dem entsprechenden `Int`–Wert erzeugt. `(ENth 0)` tritt ausschließlich in Vergleichen mit konstanten `Int` Werten auf. Der durch die Vereinfachung entstandene Vergleich zweier `Int`–Konstanten kann nun durch eine weitere Regel ebenfalls vereinfacht werden. Schließlich können dann nicht benötigte Zweige gelöscht werden.

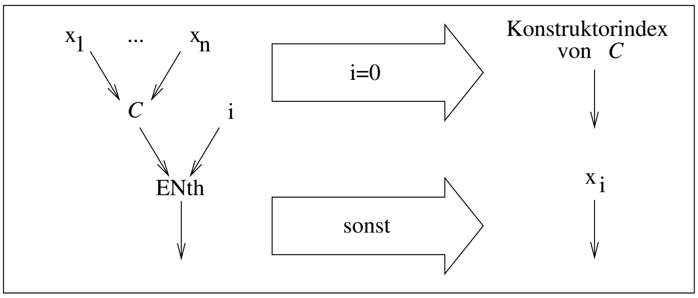

Abbildung 6.6.: Vereinfachung von Elementselektion bei Konstruktoren

6.2.3. Vereinfachung von Vergleichen

Die Anwendung eines Vergleichsoperators auf zwei identische Referenzen kann immer durch einen Wahrheitswert `True` oder `False` ersetzt werden. Abb. 6.7 zeigt diese Vereinfachung. Sie ist unabhängig vom referenzierten Ausdruck x.

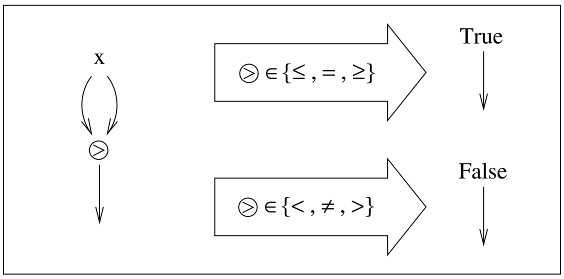

Abbildung 6.7.: Vereinfachung des Vergleichs identischer Referenzen

Eine allgemeine Aussage bei Vergleich von Ausdrücken mit unterschiedlichen Referenzen kann natürlich nicht gemacht werden. Der Vergleich zweier `Int`–Konstanten wird ebenfalls ausgewertet. Diese Vereinfachung ist nötig, um Code zu entfernen, der von nicht erreichbaren `case`–Zweigen stammt. Durch Expansion von Funktionen ist es möglich, daß das was aus einem `case`–Ausdruck geworden ist, vereinfacht werden kann, weil der Konstruktor bekannt wurde. Zusammengenommen entsprechen die Vereinfachungen von `(ENth 0)` bei bekanntem Konstruktor, `Int`–Vergleich und Löschen eines Zweiges bei bekannter Bedingung der Transformation *case-of-known-constructor* [JS98].

6.2.4. Löschen von Zweigen bei bekannter Bedingung

Beim Auftreten einer konstanten Bedingung `True` oder `False` in einer `if`–Abfrage, kann die Abfrage durch das Ergebnis des entsprechenden Zweigs ersetzt werden. Es bietet sich auch hier die Vorgehensweise an, den `if`–Ausdruck zuerst durch einen Identitätsknoten (mit Verweis zum Ergebnis des entsprechenden Zweigs) zu ersetzen. Der Identitätsknoten ist nur eine Vereinfachung in der Implementierung und wird nach dieser Transformation wieder entfernt. Er ist deshalb in Abb. 6.8 nicht dargestellt.

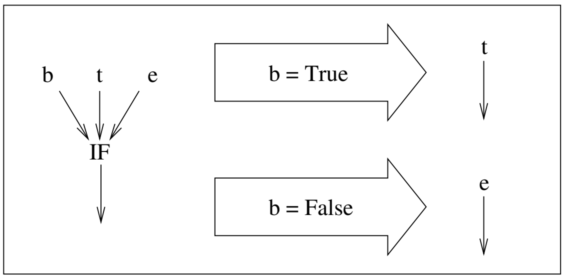

Abbildung 6.8.: Vereinfachung von `if`–Abfragen

6.3. Löschen nicht benötigter Argumente

Wird ein Funktionsargument im Funktionsrumpf nicht verwendet, so kann die Definition der Funktion verändert werden, daß dieses Argument nicht übergeben werden muß. Es müssen dann ebenfalls alle Funktionsaufrufe dieser Funktion im Programm geändert werden. Dadurch fallen Berechnungen weg, wenn das übergebene Argument im DAG mit dem Funktionsaufruf nicht mehrfach verwendet wird. Sie werden durch das Löschen von unerreichbarem Code entfernt.

Beim Löschen des toten Code können wiederum Ausdrücke gelöscht werden, die Funktionsargumente enthalten. Tritt dieser Fall ein, so werden diese Funktionsargumente nicht mehr im Rumpf verwendet und können dann in der Definition gelöscht werden. Aus diesem Grund werden die beiden Transformationen in der Optimierungsschleife in einer Subschleife durchgeführt, die wiederholt wird, falls toter Code erfolgreich entfernt wurde. Diese Schleife terminiert immer, spätestens wenn alle Funktionsargumente gelöscht wurden.

Anmerkung: Die Funktionsdefinitionen dürfen bei **OVI** nicht verändert werden, weil durch Expansion wieder ein Aufruf der Funktion mit ihrer ursprünglichen Definition in das Programm kommen kann. Damit würde ein Typkonflikt entstehen.

6.4. Löschen von unerreichbarem Code

Hierbei werden zuerst alle benötigten Knoten markiert. Es wird bei der Wurzel begonnen und alle referenzierten Teilausdrücke markiert. Von jedem markierten Ausdruck werden wiederum alle Teilausdrücke markiert. Die Knoten, die durch dieses Verfahren nicht erreicht wurden, sind toter Code und können aus dem DAG entfernt werden. Hierzu werden alle markierten Teilausdrücke mit `insert` (Abb. 5.3) in einen anfangs leeren DAG übertragen. Das Ergebnis ist ein neuer DAG, ohne unerreichbare Ausdrücke, er ersetzt den DAG der aktuellen Funktionsdefinition.

6.5. Effizienz der Optimierung

Es wird hier die Effizienz der Optimierung durch Laufzeitmessungen der in sequentiellen Code übersetzten Beispielprogramme beurteilt.

Die Messungen wurden auf einem PC (Medion, Intel PentiumII 400MHz Prozessor, 192MB Hauptspeicher) und einer Workstation (Sun UltraSPARC, 167 MHz CPU, 256MB Hauptspeicher) durchgeführt. Das von $\mathcal{HDC}$ erzeugte Ausgabeprogramm wurde auf beiden Systemen mit dem **GNU** C–Compiler, Version 2.7.2.3 mit der Option -O3 übersetzt.

Die Ergebnisse sind in Tabelle 6.9 zusammengestellt. Sie gibt die, mit dem Systemkommando „`time`“ gemessenen CPU–Zeiten von ausgewählten Beispielprogrammen in Sekunden an.

Programm	Eingabegröße	Optimierung (PC)			Optimierung (Sun)		
		0	**1**	**2**	**0**	**1**	**2**
`freqset`	0.08/100	7.2	6.0	5.9	21.7	16.1	16.2
`freqhu`	0.08/100	0.55	0.37	0.35	2.4	1.5	1.6
`pp_dc4io`	1024×1024	1.5	1.4	1.4	6.3	5.8	5.7

Abbildung 6.9.: Sequentielle Ausführungszeiten

Die Einsparungen durch die Optimierung sehen hier auf den ersten Blick nicht besonders groß aus. Bei `freqset` auf der Workstation sind es aber doch immerhin mehr als 25% Zeitersparnis im Vergleich zu dem Programm das aus nicht optimierten DAGs erzeugt wurde.

Es ist anzunehmen, daß sich die Zeiten noch verbessern, da die Speicherverwaltung des kompilierten Programms noch nicht optimal ist.

Die Programme wurden in drei Optimierungsvarianten betrachtet:

0:	Keine Elimination gegenseitiger Rekursion keine Expansion
1:	vollständige Elimination gegenseitiger Rekursion, wenn möglich durch Expansion **CVI**, Reihenfolge InlOrderIncSize
2:	Elimination gegenseitiger Rekursion wie bei **1** **OVI**

Größenbeschränkungen für die Expansion bei **1** und **2**:

Anzahl der Optimierungsschleifendurchläufe	maximal 4
Erlaubtes Funktionswachstum	maximal 4 $*$ Originalgröße
Maximale Funktionsgröße im DAG	200 Knoten

Weitere Meßwerte, bei denen Daten des Interpreters ausgewertet werden, sind im Anhang abgedruckt.

7. Abstrakte Codegenerierung

7.1. Denotationelle und operationale Semantik

Der abstrakte Code hat eine operationale Semantik. Die Ausdrücke werden, wie für den DAG, in einer Tabelle abgelegt. Sie kann durch einen Interpreter zeilenweise ausgewertet werden oder durch anschließende Codegenerierung in eine geeignete Zielsprache übersetzt werden. Beides ist nicht direkt mit DAGs möglich, die denotationell interpretiert werden.

Die denotationelle Interpretation eines DAGs geschieht auf folgende Weise: Die Auswertung beginnt mit der Wurzel des DAGs. Der dort befindliche Ausdruck ist das Ergebnis der Funktion, deren Rumpf durch den DAG definiert ist. Wenn die Auswertung für einen Teilausdruck den Wert eines anderen Knoten benötigt, so wird dieser besucht und ausgewertet. Die Knoten für `if`–Ausdrücke müssen gesondert behandelt werden: Zuerst muß der Wert der Bedingung getestet und dann der entsprechende Zweig ausgewertet werden.

Bei der operationalen Interpretation eines DAGs werden die Ausdrücke in der Tabelle nacheinander ausgewertet. Immer wenn der Wert eines Knotens in einem zusammengesetzten Ausdruck benötigt wird, so wurde dieser bereits ausgewertet. Das Ergebnis kann in dem entsprechenden Tabelleneintrag gefunden werden. Der letzte auszuwertende Ausdruck ist der im Wurzelknoten, am Ende der Tabelle. Damit ist das Ergebnis der Funktion berechnet. Bei einem DAG ohne `if`–Ausdrücke kann die Auswertung wie beschrieben erfolgen.

Ein Problem ergibt sich für `if`–Ausdrücke: Bevor die Bedingung getestet werden kann, werden bei dieser Vorgehensweise beide Zweige ausgewertet. Enthält ein Zweig einen rekursiven Aufruf, so wird der `if`–Ausdruck nie erreicht. Dieses Problem wird durch die Einführung von Knoten zur Programmablaufkontrolle gelöst.

7.2. Umwandlung in abstrakten Code

Der abstrakte Code wird aus dem DAG–Zwischencode erzeugt, indem `if`–Ausdrücke eliminiert, und drei andere Typen von Knoten eingeführt werden:

`BranchFalse` *cond i*
: Prüft den Wert der Bedingung im Knoten *cond*. Bei `True` wird die Ausführung mit dem nächsten Knoten, ansonsten mit Knoten *i* des abstrakten Codes fortgesetzt. (Abk. `BRF`)

`Jump` *i*
: Setzt die Ausführung immer mit Knoten *i* fort. (Abk. `JMP`)

`Selection` *cond a b*
: Prüft den Wert der Bedingung im Knoten *cond*. Ist dieser Wert `True`, wird der Wert von Knoten *a*, sonst der Wert von Knoten *b* zurückgegeben. (Abk. `SEL`)

Anmerkung: Die Knoten `BRF` und `JMP` sind untypisiert. Sie können nur durch die abstrakte Codegenerierung erzeugt werden. Damit ist garantiert, daß sie nicht als Teilausdrücke referenziert werden. Der Typ des Ergebnisses von `SEL` ist, wie bei `if`–Ausdrücken, identisch mit den Typen der beiden Zweige.

```
  a:                -- boolscher Wert der if-Bedingung
  ⋮
  a+k:              BRF ↑a ↑b                          k>0
  a+k+1:               -- Anfang then-Zweig
  ⋮                    -- l_t ≥ 0...Anzahl der Ausdrücke im then-Zweig
  a+k+l_t:             -- Ende then-Zweig
  a+k+l_t+1:        JMP ↑c
b=a+k+l_t+2:           -- Anfang else-Zweig
  ⋮                    -- l_e ≥ 0...Anzahl der Ausdrücke im else-Zweig
  a+k+l_t+l_e+1:       -- Ende else-Zweig
c=a+k+l_t+l_e+2:    SEL ↑a ↑t ↑e
```

Abbildung 7.1.: Struktur des abstrakten Codes für eine `if`–Abfrage

In Abb. 7.1 ist die Struktur des abstrakten Codes dargestellt, wie er für eine `if`–Abfrage erzeugt wird. Der Code für den `then`- bzw. `else`-Zweig wird durch rekursive Anwendung der abstrakten Codegenerierung für diese Zweige erzeugt. Er enthält möglicherweise weitere umgewandelte `if`–Abfragen.

Aufgrund der gemeinsamen Nutzung von Ausdrücken ist es möglich, daß `then`– oder `else`–Zweig leer sind. Sei $t < \mathtt{a+k}$, $e < \mathtt{a+k}$. Für `t` und `e` (vgl. Abb. 7.1) gilt:

$$(\mathtt{t},\mathtt{e}) = \begin{cases} (t, e) & \text{falls } \mathtt{then} \text{ leer}, \mathtt{else} \text{ leer} \\ (t, \mathtt{a+k+l_e+1}) & \text{falls } \mathtt{then} \text{ leer}, \mathtt{else} \text{ nicht leer} \\ (\mathtt{a+k+l_t}, e) & \text{falls } \mathtt{then} \text{ nicht leer}, \mathtt{else} \text{ leer} \\ (\mathtt{a+k+l_t}, \mathtt{a+k+l_t+l_e+1}) & \text{falls } \mathtt{then} \text{ nicht leer}, \mathtt{else} \text{ nicht leer} \end{cases}$$

Während der abstrakten Codegenerierung wird derzeit keine Analyse von Bedingungen durchgeführt. Verschachtelte Bedingungen, die sich gegenseitig ausschließen werden nicht vereinfacht. Es wird davon ausgegangen, daß solche Fälle entweder bereits durch die Codeoptimierung behandelt wurden oder von Laufzeitwerten abhängen.

Im abstrakten Code ist es in manchen Fällen nicht mehr sinnvoll, alle gemeinsamen Teilausdrücke zu nutzen, wie sie im DAG auftreten. Der gleiche Ausdruck kann in zwei Zweigen verwendet werden, die *möglicherweise* nicht beide in die Berechnung einfließen. Hierbei ist es wichtig, daß es in funktionalen Sprachen mehrere `if`–Ausdrücke auf der selben Ebene geben kann. `if`–Ausdrücke können wie alle anderen Ausdrücke verwendet werden, es ist also z.B. eine Verknüpfung zweier `if`–Ausdrücke mit `+` möglich. Es können also zwei unabhängige Zweige gleichzeitig benutzt werden.

Für die gemeinsame Verwendung eines gemeinsamen Teilausdrucks müßte seine Berechnung außerhalb der Zweigen erfolgen, in denen der Ausdruck vorkommt. Das wäre unter Umständen aber viel früher als der Ausdruck benötigt wird, falls er überhaupt benötigt wird. Das Vorziehen der Berechnung kostet Speicherplatz und Rechenzeit, zu einem Zeitpunkt zu dem diese Ressourcen für die wirklich benötigten Berechnungen eingesetzt werden sollten.

Andererseits darf ein Ausdruck nicht immer vorgezogen werden. Es müßten dazu die Bedingungen analysiert werden, damit keine unendlichen Rekursionen erzeugt werden.

Es gibt viele Fälle, in denen sich mit Sicherheit sagen läßt, daß der Ausdruck an höchstens einer Stelle verwendet wird, weil sich `then`- und `else`-Zweige gegenseitig ausschließen. Die Verdopplung solcher Ausdrücke führt dann nur zu einem Anwachsen des Codes aber nicht zu längeren Ausführungszeiten.

7.2.1. Hilfsdatenstruktur: Verzweigungsbaum

Die verschachtelten `if`–Ausdrücke teilen durch ihre Zweige einen DAG in Bereiche auf. Diese Struktur kann durch einen Verzweigungsbaum B dargestellt werden. Jeder seiner Knoten entspricht einem Bereich, die Wurzel dem *top-level*, auf dem die äußersten `if`–Ausdrücke liegen. Jeder `if`–Knoten des DAGs in einem Bereich führt zu zwei Unterbäumen für `then`– und `else`–Zweig.

In den Knoten von B werden die Referenzen auf alle in diesem Bereich benötigten Ausdrücke des DAGs in einer Liste abgelegt. Weitherhin wird die Referenz auf das Ergebnis des Bereichs gespeichert. Es ist auf dem *top-level* das Funktionsergebnis (Wurzel des DAGs); in jedem anderen Bereich ist es das Ergebnis des entsprechenden `then`– bzw. `else`–Zweigs.

Die Listen in B werden mit den Referenzen auf alle in einem Bereich verwendeten Ausdrücke initialisiert. Die nutzbaren Gemeinsamkeiten der dadurch entstehenden mehrfach auftretenden Referenzen werden durch die anschließende Optimierung des Verzweigungsbaums ermittelt. Es werden dazu folgende Regeln angewendet:

Horizontale Elimination gemeinsamer Referenzen
: Referenzen, die beiden Knoten eines `if`–Ausdrucks in B gemeinsam sind, werden eine Ebene nach oben, in den Vorgängerknoten v verschoben. Wird dabei eine Referenz auf einen `if`–Ausdruck verschoben, so verschieben sich auch deren Unterbäume. Die Transformation wird im Baum *bottom-up* durchgeführt. Sie beginnt mit den innersten `if`–Knoten. Auf diese Weise kann eine Referenz wiederholt nach oben verschoben werden, falls sie im zu v entgegensetzten Knoten ebenfalls auftritt.

Vertikale Elimination gemeinsamer Referenzen
: Referenzen werden in einem Knoten k gelöscht, wenn sie bereits auf dem Pfad von der Wurzel zu k liegen. In dem Bereich der durch k abgedeckt wird, kann dann auf die auf bereits vorhandene Definiton des Ausdrucks eines umschließenden Bereichs zugegriffen werden.

Sortieren
: In jedem Knoten wird die Liste der Referenzen (ausgenommen der Referenzen von `if`–Ausdrücken) in die Reihenfolge ihrer Abhängigkeiten gebracht. Das entspricht der aufsteigenden Sortierung der Referenzen. In die sortierte Liste werden `if`–Ausdrücke mit ihren den Unterbäumen an der frühestmöglichen Position eingefügt, so daß alle in den Unterbäumen benötigten Referenzen vor der Referenz des `if`–Ausdrucks stehen.

Die Optimierungen werden in dieser Reihenfolge angewendet, weil die horizontale Elimination Potential für die vertikale Elimination schaffen kann, jedoch nicht umgekehrt, da bei der letzteren nur Referenzen gelöscht werden. Sortieren wird zum Schluß durchgeführt, weil die ersten beiden Optimierungen die Reihenfolge der Abhängigkeiten nicht erhalten.

7.2.2. Erzeugung den abstrakten Codes

Nach der Optimierung des Verzweigungsbaums steht fest, welcher Ausdruck in welchem Bereich des abstrakten Codes definiert werden muß. Die abstrakte Codegenerierung durchläuft dazu den Verzweigungsbaum, legt die Kontrollknoten entsprechend der Zweige an und kopiert für jeden Bereich die Ausdrücke aus dem DAG, für die eine Referenz in der Liste im Knoten des Verzweigungsbaums steht, in die Tabelle des abstrakten Codes.

7.2.3. Beispiel

In diesem Abschnitt soll die abstrakte Codegenerierung an einem einfachen Beispiel verdeutlicht werden. Es wird im folgenden abstrakter Code für die Funktion `fib`, die in Abb. 7.2 definiert ist, erzeugt.

Ausgangspunkt für die abstrakte Codegenerierung ist die Tabelle der Ausdrücke des DAGs. Sie ist in Abb. 7.3 abgebildet.

```
fib :: Int -> Int
fib x = if x == 0
        then 1
        else if x == 1 then 1
                       else fib (x-1) + fib (x-2)
```

Abbildung 7.2.: Definition der Funktion `fib`

Die `if`–Ausdrücke teilen den DAG in 5 Bereiche ein. Sie sind in dem Verzweigungsbaum in Abb. 7.4(A) mit Kleinbuchstaben bezeichnet. Diese Bezeichnungen gelten auch für Abb. B – D. Die kursiven Zahlen in den Knoten geben die Ergebnisse des jeweiligen Bereichs an. Beispielsweise enthält der Bereich c die Referenzen auf Ausdrücke des `else`–Zweigs des äußeren `if`–Ausdrucks: das innere `if` (DAG–Referenz 11), mit den beiden Unterbäumen für seine Zweige, und die Referenzen der Ausdrücke, die zur Berechnung der Bedingung des inneren `ifs` benötigt werden. Die Referenz *11* verweist auf das Ergebnis des `else`–Zweigs des äußeren `if`–Ausdrucks (vgl. DAG, Zeile 12).

Zuerst wird die horizontale Elimination ausgeführt: Die Bereiche d und e enthalten beide die Referenz 3. Sie wird aus diesen Bereichen auf die nächsthöhere Ebene verschoben (wo sie bereits eingetragen war). Für b und c wird ebenso verfahren. Weitere horizontale Gemeinsamkeiten sind nicht vorhanden.

Es folgt die vertikale Elimination: Der Ausdruck 0 wird im abstrakten Code auf der obersten Ebene angelegt und kann daher in den Bereichen c und e verwendet werden. Andere Eliminationen sind nicht möglich.

Nach dem Sortieren hat der Verzweigungsbaum die Gestalt der Abb. 7.4(D). Diese Datenstruktur wird zur Erzeugung des abstrakten Codes verwendet. Dazu wird auf oberster Ebene begonnen und die Ausdrücke in der vorliegenden Reihenfolge aus dem DAG in den abstrakten Code kopiert. Beim Erreichen einer Verzweigung werden die benötigten Kontrollknoten angelegt und das Kopieren der Ausdrücke innerhalb der Zweige fortgesetzt.

Tabelle des DAG	Tabelle des abstr. Code
`0: x :: Int`	`0: x :: Int`
`1: 0 :: Int`	`1: 0 :: Int`
`2: (==) ↑0 ↑1 :: Bool`	`2: (==) ↑0 ↑1 :: Bool`
`3: 1 :: Int`	`3: 1 :: Int`
`4: (==) ↑0 ↑3 :: Bool`	`4: BRF ↑2 ↑6`
`5: (-) ↑0 ↑3 :: Int`	`5: JMP ↑16`
`6: fib ↑5 :: Int`	`6: (==) ↑0 ↑3 :: Bool`
`7: 2 :: Int`	`7: BRF ↑6 ↑9`
`8: (-) ↑0 ↑7 :: Int`	`8: JMP ↑15`
`9: fib ↑8 :: Int`	`9: (-) ↑0 ↑3 :: Int`
`10: (+) ↑6 ↑9 :: Int`	`10: fib ↑9 :: Int`
`11: IF ↑4 ↑3 ↑10 :: Int`	`11: 2 :: Int`
`12: IF ↑2 ↑3 ↑11 :: Int`	`12: (-) ↑0 ↑11 :: Int`
	`13: fib ↑12 :: Int`
	`14: (+) ↑10 ↑13 :: Int`
	`15: SEL ↑6 ↑3 ↑14 :: Int`
	`16: SEL ↑2 ↑3 ↑15 :: Int`

Abbildung 7.3.: DAG und abstrakter Code der Funktion `fib`

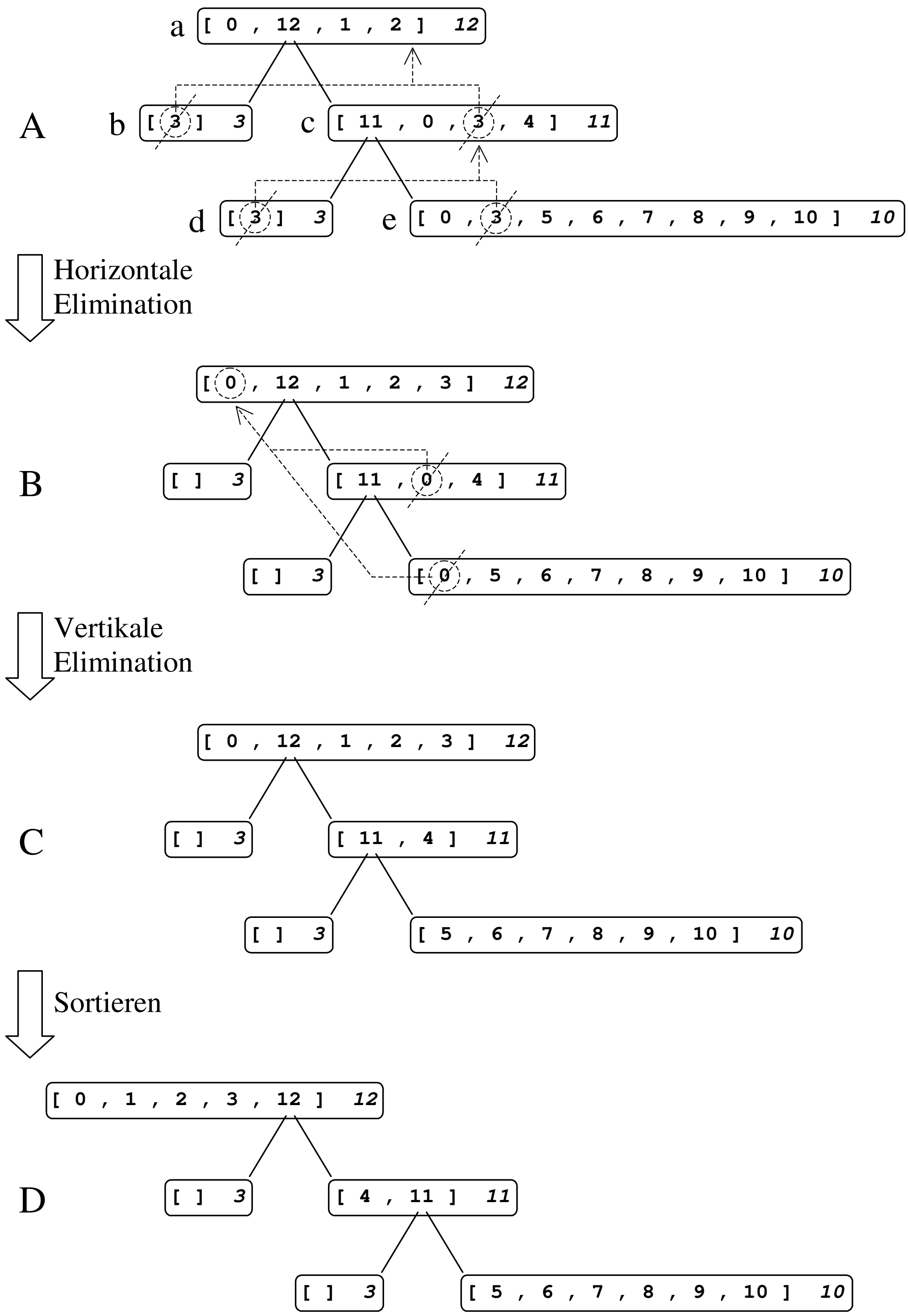

Abbildung 7.4.: Optimierung im Verzweigungsbaum

8. Ergebnisse und Ausblick

Die in dieser Arbeit vorgestellten Transformationen zur Optimierung wurden als Haskell–Funktionen implementiert und erfolgreich im $\mathcal{HDC}$–Compiler integriert.

Dieser Compiler wird am Lehrstuhl für Programmierung mit dem Ziel der Generierung paralleler Programme entwickelt. Wichtig hierbei ist auch, daß nach der Optimierung der Code möglichst schnell ausgeführt werden kann.

Für die Erzeugung von sequentiellem Code spielt die ermöglichte Nutzung gemeinsamer Teilausdrücke eine wichtige Rolle. Hierdurch kann aus dem nach der Optimierung erzeugten abstrakten Code sehr effizienter sequentieller Code erzeugt werden.

Die Expansion von Funktionsaufrufen ergibt eine zusätzliche Beschleunigung der übersetzten Programme. Allerdings kann bei sehr kleinen Beispielprogrammen in der Ausführungszeit auf einem Rechner kaum ein Unterschied zwischen Optimierung mit bzw. ohne Expansion festgestellt werden. Dies liegt möglicherweise jedoch auch an dem noch nicht optimalen Laufzeitsystem, da sich mit dem Interpreter auch für kleine Programme ein drastischer Rückgang der durchgeführten Operationen erkennen läßt.

Die Analyse der Optimierung von Beispielprogrammen für ein Datamining–Problem zeigt, daß durch die implementierte Optimierung eine große Anzahl an Operationen bei der Interpretation mit Testeingaben eingespart werden. Die erzeugten ausführbaren Programme laufen auch deutlich beschleunigt ab. Jedoch kann auch eine gute Codeoptimierung die Optimierung des, dem Programm zugrundeliegenden Algorithmus, nicht ersetzen.

Diese Arbeit liefert einen wesentlichen Beitrag zur Effizienzsteigerung des $\mathcal{HDC}$–Compilers. Trotzdem kann hiermit das Thema Codeoptimierung nicht als abgeschlossen betrachtet werden. Es gibt sehr viele weitere mögliche Transformationen mit denen eine noch bessere Vereinfachung des Zwischencodes erfolgen kann.

Es ist auch denkbar, die Expansionsbedingung durch eine „intelligentere“ Bedingung zu formulieren. Dazu sind auch die Ergebnisse der Größeninferenz interessant. Diese Phase ist jedoch derzeit noch nicht implementiert.

A. Implementierung

Die Implementierung der Codeoptimierung ist, wie alle anderen Module des $\mathcal{HDC}$–Compilers, in Haskell geschrieben. Der Compiler wurde durch das Modul `Codeopt.hs` erweitert. Einige grundlegende Definitionen wurden zur Wiederverwendung in anderen Modulen zusammen mit globalen Definitionen abgelegt. Hierunter fallen z.B. Zugriffsfunktionen auf DAGs.

`Codeopt.hs` exportiert zwei Funktionen, mit denen die Codeoptimierung vom Haupt–Modul ausgeführt werden kann:

```
codeOptimizeStopAfterDAG :: TS -> IO TS
codeOptimizeDAGs :: TS -> IO TS
```

Diese Funktionen sind `IO`–monadisch, damit Meldungen auf den Bildschirm ausgegeben werden können. Sie bekommen bei Aufruf den aktuellen Übersetzungszustand als Parameter und liefern den neuen Zustand, der die optimierten Funktionsdefinitionen enthält. Die beiden Funktionen teilen die Optimierung in zwei Teile: `codeOptimizeStopAfterDAG` endet mit der Phase der Tupelelimination. Danach liegt der DAG–Zwischencode vor. Die Optimierung muß vor Interpretation oder Codegenerierung mit `codeOptimizeDAGs` abgeschlossen werden. Diese Funktion erzeugt als letzten Schritt den abstrakten Code. Der Vorteil dieser Aufteilung ist, daß mehrere Optimierungsvarianten auf den DAG–Zwischencode angesetzt werden können, ohne wiederholtes Durchlaufen der immer gleichbleibenden Vorbereitungen auf den Hauptteil der Optimierung.

`codeOptimizeStopAfterDAG` berechnet an mehreren Stellen den Aufrufgraphen des Programms, um die Aktualität zu garantieren. Es werden weiterhin folgende Funktionen aufgerufen:

```
search_scc :: CallGraph -> [[Int]]
eliminateMutualRecursion :: CallGraph -> [[Int]] -> St ()
elimCase :: [Int] -> St ()
makeDags :: St ()
```

Sie entsprechen Phasen des Compilers (abgesehen von `search_scc`, einer Hilfsfunktion zur Suche der starken Zusammenhangskomponenten).
`eliminateMutualRecursion` benötigt den aktuellen Aufrufgraphen und das Ergebnis der Suche nach starken Zusammenhangskomponenten. Hierdurch wird die Phase der Elimination gegenseitiger Rekursion mit Hilfe der beiden vorgestellten

Verfahren implementiert. `elimCase` führt die Elimination der `case`–Ausdrücke durch. Diese Funktion erhält bei Aufruf eine Liste der benötigten Funktionen. `makeDags` wandelt den die Syntaxbaumdarstellung in den DAG–Zwischencode.

Die letztgenannten Funktionen wurden in der Zustandsmonade implementiert, wodurch das explizite Weiterreichen (*plumbing*) des Zustands entfällt.

`codeOptimizeDAGs` aktiviert die Optimierungsschleife, nach deren Verlassen nochmals versucht wird, nicht benötigte Argumente und unerreichbaren Code zu löschen. Dann werden DAGs von nicht benötigten Funktionen gelöscht. Zum Schluß wird schließlich der abstrakte Code erzeugt. Es werden hierbei folgende Funktionen verwendet:

```
inlineSimplifyLoop :: (TS,CallGraph) -> IO TS
optimizePostLoop :: TS -> IO TS
abstrCodeGen :: Fundag -> IO Fundag
```

Diese Funktionen sind wieder `IO`–monadisch geschrieben, um den Verlauf der Optimierung protokollieren zu können.

B. Meßergebnisse

In den folgenden Tabellen werden die Auswirkungen der verschiedenen Optimierungsmöglichkeiten auf Beispielprogramme . Die Tabellen wurden automatisch mit dem in Anhang C vorgestellten Verfahren ermittelt. Die in diesen Experiment nicht veränderten Optimierungsparameter, sind gleich wie in wie bei Optimierung **1** in Kapitel 6. Die Messungen wurden mit $\mathcal{HDC}$–Compiler in der Version vom 16. Juni 1999 durchgeführt.

Tabelle B.1 enthält die in Kapitel 6 versprochenen Vergleichswerte. Die folgenden Tabellen enthalten Daten für beide Verfahren zur Expansion von Funktionsaufrufen (**CVI** und **OVI**). Bei **CVI** wird die Auswirkung der verschiedenen Reihenfolgen gezeigt und mit dem Durchschnitt aus 10 Optimierungen mit einer zufälligen Reihenfolgen verglichen. Die verwendeten Eingaben sind im abgedruckten Experimentbeschreibung in Anhang B zu finden.

Die Effizienz der Optimierung wird durch zwei Prozentzahlen angegeben: Die Spalte K(%) gibt an, wie sich die Codegröße des Programms verändert hat. Es werden Anzahl der Knoten im Syntaxbaum (=100%) und Anzahl der Knoten im abstrakten Code verglichen. Hierbei ist zu beachten, daß die gemessenen Größenreduktionen nur Näherungswerte angeben, da es Unterschiede bei den Knotentypen in beiden Zwischencode–Darstellungen gibt. Unter O(%) ist ermittelt, in welchem Umfang die Anzahl der Operationen durch die Optimierung abnimmt. Die Anzahl der Operationen des ohne Expansion von Funktionen optimierten Programms wird als 100% gesetzt. Dieses Programm ist praktisch nicht optimiert, weil die meisten Vereinfachungen nur durch vorausgegangene Expansion ermöglicht werden.

In den Tabellen B.2 und B.3 wird die Optimierung des Programms von Hu [Hu99] mit und ohne Elimination gegenseitiger Rekursion gegenübergestellt. Es fällt hierbei auf, daß das Programm mit eliminierter gegenseitiger Rekursion, durch das **OVI**–Verfahren besser optimiert wird: Für die Eingabe werden weniger Operation vom Interpreter ausgeführt. Das Programm enthielt zwei Zyklen, die durch Expansion entfernt werden konnten. Es ist möglich, daß die im optimierten Programm entstandenen direkt rekursiven Funktionen durch Expansion der rekursiven Aufrufe exponentiell gewachsen sind und somit die Expansionsbedingung schnell nicht mehr erfüllbar war. Für den Interpreter wurden die Eingabedaten aus Kapitel 6 verwendet.

Tabelle B.4 enthält die Daten für das Quicksort–Programm und Tabelle B.5 zeigt die Ergebnisse für pythagoräische Zahlentripel–Programm.

	Programmgröße			Interpreter–Statistik		
		DAG	ACode			
	Fkt	Kn	Kn	Op	Sched	Par
`freqset`						
InlOrderCallGraph	27	412	493	31727	587	54.0
InlOrderFunNumber	27	410	491	31685	547	57.9
InlOrderIncSize	25	377	458	31703	552	57.4
InlOrderDecSize	29	476	579	31727	587	54.0
InlOrderRandom ∅	28.3	442.6	534.8	31701.6	574.4	55.2
InlOrderRandom 0	30	485	588	31731	589	53.9
InlOrderRandom 1	28	420	501	31731	580	54.7
InlOrderRandom 2	28	447	550	31707	567	55.9
InlOrderRandom 3	27	411	492	31727	587	54.0
InlOrderRandom 4	29	475	579	31727	578	54.9
InlOrderRandom 5	28	414	495	31662	560	56.5
InlOrderRandom 6	31	485	588	31710	565	56.1
InlOrderRandom 7	28	414	495	31663	561	56.4
InlOrderRandom 8	28	470	574	31679	583	54.3
InlOrderRandom 9	26	405	486	31679	574	55.2
`freqset_rek`						
InlOrderCallGraph	51	1152	1474	45824	4966	9.2
InlOrderFunNumber	57	1164	1501	39957	4632	8.6
InlOrderIncSize	54	1068	1380	39465	4632	8.5
InlOrderDecSize	56	1223	1578	47219	4835	9.8
InlOrderRandom ∅	52.5	1141.3	1533.1	42474.0	4433.9	9.7
InlOrderRandom 0	52	1170	1479	46612	4788	9.7
InlOrderRandom 1	50	1125	1486	38582	4701	8.2
InlOrderRandom 2	52	1200	1531	43341	3859	11.2
InlOrderRandom 3	52	1144	1491	40192	4782	8.4
InlOrderRandom 4	53	1129	1450	39851	4581	8.7
InlOrderRandom 5	48	1070	1561	44289	3620	12.2
InlOrderRandom 6	54	1124	1602	42927	4856	8.8
InlOrderRandom 7	56	1179	1553	38276	4459	8.6
InlOrderRandom 8	55	1159	1693	45746	3752	12.2
InlOrderRandom 9	53	1113	1485	44924	4941	9.1

Abbildung B.1.: Vergleich verschiedener Reihenfolgen bei der Expansion

Programm: `freqhu.hdc`, Ohne Elimination gegenseitiger Rekursion Größe vor Optimierung: 46 Funktionen, 431 Knoten DAGs ohne Optimierung: 37 Funktionen, 186 Knoten								
Expansion	Programmgröße				Interpreter–			
O=OVI		DAG	ACode		Statistik			
C=CVI	Fkt	Kn	Kn	K%	Op	O%	Sched	Par
Keine	37	185	198	45.9	5800	100.0	509	11.4
O	18	239	296	68.7	3914	67.5	439	8.9
C, CallGraph	10	187	250	58.0	3398	58.6	386	8.8
C, TopSort	10	187	250	58.0	3398	58.6	386	8.8
C, FunNumber	12	211	275	63.8	3402	58.7	386	8.8
C, IncSize	10	165	219	50.8	3397	58.6	385	8.8
C, DecSize	10	178	239	55.5	3398	58.6	386	8.8
C, Random ∅	10.2	181.5	241.2	56.0	3403.0	58.7	383.6	8.9

Abbildung B.2.: Optimierungen im Vergleich: `freqhu` - 1

Programm: `freqhu.hdc`, Mit Elimination gegenseitiger Rekursion Größe vor Optimierung: 46 Funktionen, 431 Knoten DAGs ohne Optimierung: 37 Funktionen, 188 Knoten								
Expansion	Programmgröße				Interpreter–			
O=OVI		DAG	ACode		Statistik			
C=CVI	Fkt	Kn	Kn	K%	Op	O%	Sched	Par
Keine	37	187	200	46.4	5483	100.0	492	11.1
O	14	249	326	75.6	3094	56.4	412	7.5
C, CallGraph	9	173	234	54.3	3397	62.0	385	8.8
C, TopSort	9	173	234	54.3	3397	62.0	385	8.8
C, FunNumber	11	193	252	58.5	3402	62.0	391	8.7
C, IncSize	10	194	255	59.2	3410	62.2	366	9.3
C, DecSize	10	178	239	55.5	3398	62.0	386	8.8
C, Random ∅	10.0	189.5	249.1	57.8	3402.8	62.1	381.9	8.9

Abbildung B.3.: Optimierungen im Vergleich: `freqhu` - 2

Programm: `quicksort.hdc` Größe vor Optimierung: 29 Funktionen, 243 Knoten DAGs ohne Optimierung: 24 Funktionen, 123 Knoten								
Expansion	Programmgröße				Interpreter–Statistik			
O=OVI		DAG	ACode					
C=CVI	Fkt	Kn	Kn	K%	Op	O%	Sched	Par
Keine	24	113	117	48.1	2087	100.0	405	5.2
					1116	100.0	171	6.5
O	9	67	71	29.2	1252	60.0	236	5.3
					666	59.7	98	6.8
C, CallGraph	8	61	65	26.7	1251	59.9	235	5.3
					665	59.6	97	6.9
C, TopSort	8	61	65	26.7	1251	59.9	235	5.3
					665	59.6	97	6.9
C, FunNumber	9	67	71	29.2	1252	60.0	236	5.3
					666	59.7	98	6.8
C, IncSize	8	61	65	26.7	1251	59.9	235	5.3
					665	59.6	97	6.9
C, DecSize	8	61	65	26.7	1251	59.9	235	5.3
					665	59.6	97	6.9
C, Random ∅	8.1	61.6	65.6	27.0	665.1	59.6	97.1	6.8

Abbildung B.4.: Optimierungen im Vergleich: `quicksort`

Programm: `tups.hdc` Größe vor Optimierung: 20 Funktionen, 205 Knoten DAGs ohne Optimierung: 15 Funktionen, 103 Knoten								
Expansion	Programmgröße				Interpreter–Statistik			
O=OVI		DAG	ACode					
C=CVI	Fkt	Kn	Kn	K%	Op	O%	Sched	Par
Keine	15	103	111	54.1	55329	100.0	283	195.5
O	6	118	136	66.3	42176	76.2	253	166.7
C, CallGraph	6	107	123	60.0	42176	76.2	253	166.7
C, TopSort	6	107	123	60.0	42176	76.2	253	166.7
C, FunNumber	6	107	123	60.0	42176	76.2	253	166.7
C, IncSize	6	107	123	60.0	42176	76.2	253	166.7
C, DecSize	6	107	123	60.0	42176	76.2	253	166.7
C, Random ∅	6.0	107.0	123.0	60.0	42176.0	76.2	253.0	166.7

Abbildung B.5.: Optimierungen im Vergleich: `tups`

C. Automatische Ermittlung von Meßergebnissen

C.1. Experiment–Schnittstelle

Das $\mathcal{HDC}$–System hat eine Schnittstelle für die Kontrolle der Durchführung von Experimenten und für das automatische Aufzeichnen von Experimentdaten. Der Benutzer kann im $\mathcal{HDC}$–Modul `Experiment.hs` durch die eigene Definition eines Protokolls Parametersätze für Optimierungen, sowie Eingabe- und Ergebnisdaten mit dem Haupt–Modul austauschen. Das Hauptprogramm übernimmt den Aufruf der Optimierungen mit den gewünschten Parametern und führt den Interpreter auf einer Menge von Eingabedaten aus.

Der Benutzer kann festlegen, welche Daten gespeichert werden sollen und wie diese aufbereitet werden (z.B. als LaTeX–Tabelle). `Experiment.hs` wird zusammen mit den anderen Modulen von $\mathcal{HDC}$ übersetzt. Das Experiment–Modul muß passend für das $\mathcal{HDC}$–Programm geschrieben sein, mit dem experimentiert werden soll. Insbesondere muß es dem Interpreter Eingaben im richtigen Format liefern. Eine Veränderung des Experiments macht eine Neuübersetzung der davon abhängigen Module (Interpreter– und Haupt–Modul) erforderlich.

Die Anbindung erfolgt über `IO`–monadische Funktionen, damit Informationen auf den Bildschirm und in Dateien ausgegeben werden können. Es müssen folgende experimentspezifische Funktionen definiert werden:

```
initExp :: TS -> IO (TS, ExperimentProtocol)

getNextParams :: (TS, ExperimentProtocol)
                 -> IO (TS, ExperimentProtocol,
                        Maybe ExperimentParams)

getInput :: (TS, ExperimentProtocol)
            -> ExperimentResults
            -> IO (TS, ExperimentProtocol, Maybe String)

formatResults :: (TS, ExperimentProtocol) -> IO ()
```

`TS` ist der Typ des globalen Zustands, in dem wichtige Informationen des Übersetzungszustands abgelegt sind. Hier befinden sich unter anderem die top–level Funktionsdefinitionen, Warnungen und Fehlermeldungen und der vom Interpreter berechnete Schedule. Die meisten Felder in `TS` sind irrelevant für die Experimentbeschreibung und sollten nicht verändert werden, um die Integrität des Compilers zu erhalten. Die Rückgabe eines Zustands ist nur für das Eintragen von Fehlermeldungen bestimmt.

Von dem Haupt–Modul wird zur Initialisierung einer Experimentserie die Funktion `initExp` aufgerufen. Das Hauptprogramm erfragt dann wiederholt neue Experimentparameter durch `getNextParams`. Sie enthalten die Einstellungen, mit denen die DAGs optimiert werden. Die Optimierung wird immer auf die erste Version der DAGs (nach abgeschlossener Tupelelimination) angesetzt. Die Phasen davor (Parser, Typechecker, Elimination funktionaler Argumente und Aufbau unoptimierter DAGs) werden nicht mehr durchlaufen, wodurch bei größeren Programmen erheblich Zeit einspart wird.

Die Experimentserie ist beendet, sobald `getNextParams` keinen Parametersatz (`Nothing`) mehr zurückgibt.

Jedes optimierte Programm kann für mehrere Eingabedaten interpretiert werden. Die Funktion `getInput` wird vom Hauptprogramm zur Anforderung von Daten aufgerufen. Auch hier wird das Ende durch den Rückgabewert `Nothing` erzwungen. `getInput` erhält als Eingabe das Ergebnis der letzten Interpretation. Es ist nach jeder neuen Optimierung beim ersten Aufruf von `getInput` leer. Wenn das Ergebnis nicht aus einer Fehlermeldung besteht, können nun die Interpreterdaten aus dem übergebenen Zustand analysiert und im Protokoll aufgenommen werden.

Das nachfolgend abgedruckte Experiment–Modul dient als Beleg dafür, wie die Tabellen im Anhang B zustande gekommen sind. Es zeigt auch wie das Protokoll verwaltet werden kann.

C.2. Experiment.hs

```
module Experiment
  (ExperimentProtocol, ExperimentParams, ExperimentResults,
   initExp, getNextParams, getInput, formatResults,
   optimOpt, statistics) where

import Types
import List

-- For use with quicksort, tups, freqset, freqhu
-- inputs must be given in table 'inputdata'. there are at least
-- the number of 'inputs' required. the inputs are looked up by
-- "'program'.'n'" where 'n' ranges from 1 to 'inputs'.

---------------------------------------------------------------------
-- Configuration

program = "tups"
comment = ""
inputs  = 1

inputdata = [
  ("quicksort.1", show [10,9..0]),
  ("quicksort.2", show [4,0,5,3,8,10,3,1,10,6,2]),
  ("tups.1", "30"),
  ("freqset.1", "0.3 "++show bag),
  ("freqhu.1", "0.3 "++show [0..6]++" "++show bag)
 ]

bag = [[2,3],[2,3,4],[2,3,4,5],[2,3,4,5,6],[3,4],[3,4,5],[4,5,6]]


initAll   = initOptimizeOptions{
              inlOn=True, inlAlgo=0, inlLoops=4,
              inlMaxRelFunSize=4, inlMaxAbsFunSize=200 }
initNOI   = initEP initAll{ inlOn=False }
initOVI   = initEP initAll{ inlAlgo=1 }
initCVI n = initEP initAll{ inlOrder=toEnum n }
initRan   = initEP initAll{ inlOrder=InlOrderRandom }
---------------------------------------------------------------------

data SingleExperiment =
  SingleExp {
    expCount :: Int, expInput :: String, expResult :: String,
    expNoRed :: Int, expAvePar :: Double, expSched :: Int,
    optInfo :: OptimizeInfo
  } deriving (Eq,Show,Read)

initSingleExp = SingleExp
  { expCount=0, expInput="", expResult="", expNoRed=0,
    expAvePar=0.0, expSched=0, optInfo=initOptimizeInfo }
```

```
data ExperimentParams =
  ExperimentParams {
    optimOpt :: OptimizeOptions
  } deriving (Show,Read,Eq)

initEP o = ExperimentParams { optimOpt = o }

type ExperimentProtocol = [(ExperimentParams,[SingleExperiment])]
type ExperimentResults  = Either String String

initExp :: TS -> IO (TS,ExperimentProtocol)
initExp st = return (st,[])

getNextParams :: (TS,ExperimentProtocol)
               -> IO (TS,ExperimentProtocol,Maybe ExperimentParams)
getNextParams (st,protocol)
 = if null protocol
   then
     do putStrLn "1. Optimization"
        return (st, [(initEP initOptimizeOptions,[])], Just initNOI)
   else
     let l = length protocol-1 in
     if l < 6+11
     then
       do putStrLn (show (l+2)++". Optimization")
          if l < 6
           then
             do let newparams = if l==0 then initOVI else initCVI(l-1)
                return (st, (newparams,[]):protocol, Just newparams)
           else
             do let newparams = initRan
                return (st, (newparams,[]):protocol, Just newparams)
     else
       do putStrLn "end of experiments"
          return (st,protocol,Nothing)

getInputValue :: Int -> String
getInputValue n
 = let s = program ++ "." ++ show n in
   case lookup s inputdata of
     Just input -> input
     Nothing -> error ("Experiment.getInputValue: " ++ s
                      ++ ": input not found")

getInput :: (TS,ExperimentProtocol) -> ExperimentResults
            -> IO (TS,ExperimentProtocol,Maybe String)
getInput (st,protocol@((params,singleexp):ps)) eitherresults
 =  case eitherresults of
      Left error ->
        do putStrLn "Error occured in experiment:"
           putStrLn error
           return (st,protocol,Nothing)
```

```
      Right "" ->
        do let input = getInputValue 1
           putStrLn ("first input: " ++ show input)
           return (st, (params, initSingleExp{expInput=input}
                        : singleexp):ps,
                        Just input)
      Right result ->
        do let lastexp = head singleexp
               n = expCount lastexp + 1
           if n < inputs
            then
             do let input = getInputValue (n+1)
                putStrLn ("next input: "++ show input)
                let newprotocol
                      = (params, initSingleExp{expInput=input,expCount=n}
                                 : myExpResult (st,lastexp) result
                                 : tail singleexp)
                        : ps
                return (st, newprotocol, Just input)
            else
             do putStrLn "input set finished"
                let newprotocol
                      = (params, myExpResult (st,lastexp) result
                                 : tail singleexp)
                        : ps
                return (st, newprotocol, Nothing)

formatResults :: (TS,ExperimentProtocol) -> IO ()
formatResults (st,protocol)
 = do let results = make_TeX_table protocol
      writeFile "results.tex" results


statistics :: TS -> (Maybe (Double,Double,Int,Int),OptimizeInfo)
statistics st
 = let (sumsqrs,sumlin,n)
         = strict3 avl_reduce (\(sumsqrs,sumlin,n) v ->
             let a = sumsqrs+(snd v)*(snd v)
                 b = sumlin+snd v
                 c = n+1
             in seq a (seq b (seq c (a,b,c)))) (0,0,0) (schedule st)
       avg
         = if n /= 0 then fromInt sumlin / fromInt n else 0.0
       sigma
         = let divisor = n * (n-1) in
           if divisor == 0
           then 0.0
           else sqrt (fromInt (n * sumsqrs - sumlin*sumlin)
            / fromInt divisor)
   in
   if n == 0
   then (Nothing,optimizeInfo st)
   else (Just (avg,sigma,sumlin,n),optimizeInfo st)
```

```
myExpResult :: (TS,SingleExperiment) -> String -> SingleExperiment
myExpResult (st,lastexp) result =
 case statistics st of
   (Nothing,info) ->
     lastexp { expNoRed=0, expAvePar=0.0, expSched=0,
               expResult=result, optInfo=info }
   (Just (avg,sigma,sumlin,n),info) ->
     lastexp { expNoRed=sumlin, expAvePar=avg, expSched=n,
               expResult=result, optInfo=info }
```

```
rnd :: Double -> Int -> Double
rnd d n = fromInt (floor (d * 10^n + 0.5)) / 10^n

showOptInfo f s = (show . f . optInfo) s
showExpData f s = (show . f) s
table_format = "|l||c|c|c|c||c|c|c|c|"
title_line progname ((_,se:_):_)
  = let f  = showOptInfo funCountTree se
        k  = showOptInfo sizeTree se
        fd = showOptInfo funCountDag se
        kd = showOptInfo sizeDag se
    in
     "\\multicolumn{9}{|l|}{Programm: \\V{"++progname++"} " ++
     comment ++ "}\\\\\n" ++
     "\\multicolumn{9}{|l|}{Gr\"o\"se vor Optimierung: " ++
     f ++ " Funktionen, " ++ k ++ " Knoten}\\\\\n" ++
     "\\multicolumn{9}{|l|}{DAGs ohne Optimierung: " ++
     fd ++ " Funktionen, " ++ kd ++ " Knoten}\\\\\n" ++
     "\\hline\n" ++
     "\\hline\n" ++
     "Expansion& \\multicolumn{4}{c||}{Programmgr\"o\"se}\n" ++
     "&\\multicolumn{4}{c|}{Interpreter--}\\\\\n" ++
     "{\\bf O}={\\bf OVI}& & DAG & ACode & "++
     "& \\multicolumn{4}{c|}{Statistik}\\\\\n" ++
     "{\\bf C}={\\bf CVI} "++
     "& Fkt & Kn & Kn & K\\% & Op & O\\% & Sched & Par\\\\\n"
table_lines ps =
  let -- results from InlOrderRandom experiments
      xs = map (head . snd) (drop 7 ps)
      -- compute (rounded) average for a field
      avg f  = sum (map f xs) / fromInt (length ps - 7)
      savg f = show (rnd (avg f) 1)
      -- get number of operations from experiment without
      -- inline expansion (for percentage calculation)
      ((_,s):_) = ps
      red_noexp = map expNoRed (reverse s)
      -- format interpreter result columns
      irs (n,se)
       = (if n>1 then "&&&&" else "")
```

```
            ++ "& "++showExpData expNoRed se
            ++ "& "++show (rnd (
                       fromInt (100*expNoRed se)
                       / fromInt (red_noexp!!(n-1))) 1)
            ++ "& "++showExpData expSched se
            ++ "& "++show (rnd (expAvePar se) 1)
            ++ "\\\\\n"
       ls
        = [ case n of
              -7 -> "Keine"
              -6 -> "{\\bf O}"
              _  -> "{\\bf C}, "++drop 8 (show ((inlOrder . optimOpt) ep))
            ++ (if n >= 0 then " " ++ show n else "")
            ++ "& "++showOptInfo funCountOptimizedDag se
            ++ "& "++showOptInfo sizeOptimizedDag se
            ++ "& "++showOptInfo sizeAbstractCode se
            ++ "& "++show (rnd (
                       fromInt (100*sizeAbstractCode (optInfo se))
                       / fromInt (sizeTree (optInfo se))) 1)
            ++ concatMap irs (zip [1..] (reverse es))
          | (n,(ep,es@(se:_))) <- (zip [-7..] ps) ]
   in concat $
        take 7 ls
        ++ ["\\hline\n"]
        ++ ["{\\bf C}, Random $\\emptyset$ "
             ++ "& "++savg (fromInt . funCountOptimizedDag . optInfo)
             ++ "& "++savg (fromInt . sizeOptimizedDag . optInfo)
             ++ "& "++savg (fromInt . sizeAbstractCode . optInfo)
             ++ "& "++show (rnd (
                        100*avg (fromInt . sizeAbstractCode . optInfo)
                         / fromInt (sizeTree (optInfo (last s)))) 1)
             ++ "& "++savg (fromInt . expNoRed)
             ++ "& "++show (rnd (
                        100*avg (fromInt . expNoRed)
                         / fromInt (last red_noexp)) 1)
             ++ "& "++savg (fromInt . expSched)
             ++ "& "++savg (read . show . expAvePar)
             ++ "\\\\\n"]

make_TeX_table :: ExperimentProtocol -> String
make_TeX_table p' = let p = reverse (tail p') in
 "\\begin{figure}[h]\n" ++
 "\\begin{center}\\begin{tabular}{"++ table_format ++"}\n" ++
 "\\hline\n" ++
 title_line (program++".hdc") p ++
 "\\hline\n" ++
 table_lines p ++
 "\\hline\n" ++
 "\\end{tabular}\\end{center}\n" ++
 "\\caption{Optimierungen im Vergleich: \\V{"++program++"}}\n" ++
 "\\label{tab:cmpopt-"++program++"}\n" ++
 "\\end{figure}\n"
```

D. $\mathcal{HDC}$–Beispielprogramme

Die im folgenden angegebenen Beispielprogramme wurden für die Messungen in dieser Arbeit verwendet.

`freqset.hdc`
: Dieses Programm stammt aus [HLG+99]. Es wird dort beschrieben. Es implementiert einen Datamining–Algorithmus, der nach häufig auftretenden Mengen sucht.

`freqset_rek.hdc`
: Noch eine Implementierung von `freqset`. Es werden Skelette nur für die Ein-/Ausgabe verwendet. Alle anderen Skelette wurden durch rekursive Funktionen ersetzt. Dieses „schlechte“ $\mathcal{HDC}$–Programm dient nur für Vergleichsdaten.

`freqhu.hdc`
: Effizienterer Algorithmus von Zhenjiang Hu für das Problem der häufig auftretenden Mengen. [Hu99] Ein–/Ausgabe unterscheiden sich leicht von den `freqset`–Programmen.
Mit der freundlichen Genehmigung des Autors konnte dieses Programm hier ebenfalls abgedruckt werden.

`pp_dc4io.hdc`
: Berechnet das Produkt zweier Polynome nach Algorithmus von Karatsuba.

`quicksort.hdc`
: Sortiert eine Liste von Zahlen mit dem bekannten Algorithmus, hier unter Verwendung des `dc0`–Skeletts.

`tups.hdc`
: Sucht *pythagoräische Zahlentripel* (a,b,c), bei denen für natürliche Zahlen a, b, c die Gleichung $c^2 = a^2 + b^2$ erfüllt ist.

D.1. `freqset.hdc`

```
import HDCPrelude

compareSet :: Ord a => [a] -> [a] -> Int
-- compares two ordered sets with respect to first the size, then the
-- lexicographic ordering, delivers -1 if xs<ys, 0 if xs=ys, 1 if xs>ys
compareSet xs ys
 = if length xs == length ys
      then let firstdiff =
                 skel_while (\i -> if (i<length xs) then (xs!!i==ys!!i)
                                                    else False ) (+1) 0
           in if firstdiff == length xs
                 then 0
                 else if xs!!firstdiff > ys!!firstdiff then 1 else (-1)
      else if length xs > length ys then 1 else (-1)

isElem :: Ord a => a -> [a] -> Bool
-- checks if e is element of the set s
isElem e s = any (==e) s

isSubSet :: Ord a => [a] -> [a] -> Bool
-- checks if sub is a subset of super
isSubSet sub super = all (\s -> isElem s super) sub


insertSet :: Ord a => a -> [a] -> [a]
-- adds x to the set xs
insertSet x xs = filter (<x) xs ++ (x : filter (>x) xs)

remDuplicates :: Ord a => [[a]] -> [[a]]
-- removes all duplicates in a set of sets
remDuplicates
 = let pivot xs = xs!!(length xs ‘div‘ 2)
       p xs = length xs < 2
       b xs = xs
       d xs = let less
                   = filter (\x -> compareSet x (pivot xs) == (-1)) xs
                  greater
                   = filter (\x -> compareSet x (pivot xs) == 1)    xs
              in [less,greater]
       c xs [as,bs] = as ++ (pivot xs : bs)
   in dc0 p b d c

countSubsets :: [Int] -> [[Int]] -> Int
-- counts the number of elements in bs which b is a subset of
countSubsets b bs = length (filter (isSubSet b) bs)

fracOK :: [[Int]] -> Double -> [Int] -> Bool
-- given a bag of sets bs, a fraction f and a set b
-- tells if b occurs as a subset in more than the fraction f of
--       all elements of bs
```

```
fracOK bs f b = ( fromInt (countSubsets b bs) / fromInt (length bs)) > f

freqSets :: [[Int]] -> Double -> [Int] ->
            ([[Int]],[[Int]]) -> Int -> ([[Int]],[[Int]])
-- given
-- bbag: the bag of sets
-- frac: the fraction of occurrences required
-- sngl: all singleton frequent sets
-- (f,filast): the current/previous collection of frequent sets
-- i: the current level
freqSets bbag frac sngl (f,filast) i
 = (\fi -> (f++fi,fi))
   (filter (fracOK bbag frac)
           (remDuplicates
               (filter (\xs -> length xs == i)
                       [ insertSet s1 s2 | s1 <- sngl, s2 <- filast])))

datamineSet :: Int -> Int -> [[Int]] -> Double -> [[Int]]
-- data mining on the Set representation to enable
-- efficient parallel subset testing
datamineSet m u bbag frac
 =  (\siOK ->
     fst (foldl (freqSets bbag frac (map (!!0) siOK))
                (siOK,siOK) [2..u]))
    (map (\x->[x]) (filter (\x -> fracOK bbag frac [x]) [0..m]))
-- list of single items that satisfy fraction condition

datamine :: [[Int]] -> Double -> [[Int]]
-- the entire datamining algorithm
datamine bag frac =
  let maxi = red max 0 (concat bag)
      ubnd = red max 0 (map length bag)
  in  datamineSet maxi ubnd bag frac

parmain :: IO Unit
parmain = get >>= \threshold ->
          get >>= \bag ->
          put (datamine bag threshold)
```

D.2. freqset_rek.hdc

```
import HDCPrelude

filter_rek :: (a->Bool) -> [a] -> [a]
filter_rek p [] = []
filter_rek p (x:xs) = (if p x then [x] else []) ++ filter_rek p xs

dc0_rek :: (a->Bool) -> (a->b) -> (a->[a]) -> (a->[b]->b) -> a -> b
dc0_rek p b d c x
 = if p x then b x else c x (map_rek (dc0_rek p b d c) (d x))

while_rek :: (a->Bool) -> (a->a) -> a -> a
while_rek p f x = if p x then while_rek p f (f x) else x

map_rek f []     = []
map_rek f (x:xs) = f x : map_rek f xs

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

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

any_, all_ :: (a -> Bool) -> [a] -> Bool
any_ p = foldr_rek (||) False . map_rek p
all_ p = foldr_rek (&&) True . map_rek p

concat_ :: [[a]] -> [a]
concat_ xs = foldl_rek (++) [] xs

concatMap_ :: (a -> [b]) -> [a] -> [b]
concatMap_ f = concat_ . map_rek f

compareSet :: Ord a => [a] -> [a] -> Int
compareSet xs ys
 = if length xs == length ys
      then let firstdiff =
                while_rek (\i -> if (i<length xs) then (xs!!i==ys!!i)
                                                   else False ) (+1) 0
           in if firstdiff == length xs
                 then 0
                 else if xs!!firstdiff > ys!!firstdiff then 1 else (-1)
      else if length xs > length ys then 1 else (-1)

isElem :: Ord a => a -> [a] -> Bool
isElem e s = any_ (==e) s

isSubSet :: Ord a => [a] -> [a] -> Bool
isSubSet sub super = all_ (\s -> isElem s super) sub
```

```
insertSet :: Ord a => a -> [a] -> [a]
insertSet x xs = filter_rek (<x) xs ++ (x : filter_rek (>x) xs)

remDuplicates :: Ord a => [[a]] -> [[a]]
remDuplicates
 = let pivot xs = xs!!(length xs `div` 2)
       p xs = length xs < 2
       b xs = xs
       d xs =
         let le = filter_rek (\x -> compareSet x (pivot xs) == (-1)) xs
             gr = filter_rek (\x -> compareSet x (pivot xs) == 1)    xs
         in [le,gr]
       c xs [as,bs] = as ++ (pivot xs : bs)
   in dc0_rek p b d c

countSubsets :: [Int] -> [[Int]] -> Int
countSubsets b bs = length (filter_rek (isSubSet b) bs)

fracOK :: [[Int]] -> Double -> [Int] -> Bool
fracOK bs f b = (fromInt (countSubsets b bs) / fromInt (length bs)) > f

freqSets :: [[Int]] -> Double -> [Int] ->
            ([[Int]],[[Int]]) -> Int -> ([[Int]],[[Int]])
freqSets bbag frac sngl (f,filast) i
 = (\fi -> (f++fi,fi))
   (filter_rek (fracOK bbag frac)
               (remDuplicates
                   (filter_rek (\xs -> length xs == i)
                      (let ok1 s1 = let ok2 s2 = [ insertSet s1 s2 ]
                                    in  concatMap_ ok2 filast
                       in  concatMap_ ok1 sngl))))

datamineSet :: Int -> Int -> [[Int]] -> Double -> [[Int]]
datamineSet m u bbag frac
 = (\siOK ->
    fst (foldl_rek (freqSets bbag frac (map_rek (!!0) siOK))
                   (siOK,siOK) [2..u]))
   (map_rek (\x->[x]) (filter_rek (\x -> fracOK bbag frac [x]) [0..m]))

datamine :: [[Int]] -> Double -> [[Int]]
datamine bag frac =
  let maxi = foldl_rek max 0 (concat_ bag)
      ubnd = foldl_rek max 0 (map_rek length bag)
  in  datamineSet maxi ubnd bag frac

parmain :: IO Unit
parmain = get >>= \threshold ->
          get >>= \bag ->
          put (datamine bag threshold)
```

D.3. freqhu.hdc

```
-- Copyright by Zhenjiang Hu (author)
-- Published here with the permission of the author.


left :: [a] -> [a]
left xs =  take (length xs `div` 2) xs

right :: [a] -> [a]
right xs =  drop (length xs `div` 2) xs



parmain :: IO Unit
parmain = get >>= \threshold ->
          get >>= \elements ->
          get >>= \bag ->
          put (fs (floor (fromInt (length bag) * threshold)) elements bag)

fs :: Int -> [Int] -> [[Int]] -> [[Int]]
fs least ms nss = fs1 least ms nss []

fs1 :: Int -> [Int] -> [[Int]] -> [Int] -> [[Int]]
fs1 least ms nss r = tup least ms [(r,nss)]

tup :: Int -> [Int] -> [([Int],[[Int]])] -> [[Int]]
tup least [] rs = map fst rs
tup least (m:ms) rs = tup least ms (add least m rs)

add :: Int -> Int -> [([Int],[[Int]])] -> [([Int],[[Int]])]
add least m rs
 = if length rs==1
    then let (r,nss) = rs!!0
             nss1 = filter (m `isElem`) nss
         in if length nss1 > least
               then (m:r,nss1):rs
               else rs
    else add least m (left rs) ++ add least m (right rs)

isElem :: Int -> [Int] -> Bool
isElem e [] = False
isElem e (x:xs) = if e<x then False
                         else if e==x then True
                                      else isElem e xs
```

D.4. pp_dc4io.hdc

```
import HDCPrelude

infixl 2 ->>

-- polynomial product
-- the length must be a power of 2
-- using Karatsuba's algorithm

karatsuba :: [Int] -> [Int] -> [Int]
karatsuba a b =
  let basic (x,y) = (0,x*y)
      divide  [(xh,yh),(xl,yl)]
            = [(xh,yh),(xl,yl),(xh+xl,yh+yl)]
      combine [(hh,hl),(lh,ll),(mh,ml)]
            = [(hh,lh+ml-hl-ll),(hl+mh-hh-lh,ll)]
  in ilog2 (length a)                     ->> \n ->
     zip a b                              ->> \x ->
     dc4io 3 2 2 basic divide combine n x ->> \z ->
     map fst z ++ map snd z

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

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

parmain :: IO Unit
parmain = get >>= \a ->
          get >>= \b ->
          put ((karatsuba a b) :: [Int])
```

D.5. quicksort.hdc

```
qs :: [Int] -> [Int]
qs ys = let p xs = length xs < 2
            b xs = xs
            pivot xs = xs!!0
            d xs = [filter (< pivot xs) xs, filter (> pivot xs) xs]
            c xs [ys,zs] = ys ++ (filter (== pivot xs) xs) ++ zs
        in dc0 p b d c ys

parmain :: IO Unit
parmain = get >>= \x ->
          put (qs (x::[Int]))
```

D.6. tups.hdc

```
tups :: Int -> [(Int,Int,Int)]
tups n
 = [ (a,b,c) | a<-[1..n], b<-[a+1..n], c<-[b+1..n], c*c==a*a+b*b ]

tolist :: [(Int,Int,Int)]->[Int]
tolist [] = []
tolist ((a,b,c):xs) = [a,b,c,0] ++ tolist xs

parmain :: IO Unit
parmain
 = get >>= \n ->
   put (tolist (tups n))
```

Literaturverzeichnis

[AHU74] Alfred V. Aho, John E. Hopcroft, und Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, 1974.

[AHU87] Alfred V. Aho, John E. Hopcroft, und Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, Inc., 1987.

[ASU86] Alfred V. Aho, Ravi Sethi, und Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley series in Computer Science, 1986.

[AU96] Alfred V. Aho und Jeffrey D. Ullman. *Informatik: Datenstrukturen und Konzepte der Abstraktion*. International Thomson Publishing, 1996.

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

[CLR90] Thomas H. Cormen, Charles E. Leiserson, und Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

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

[GLS94] William Gropp, Ewing Lusk, und Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation Series. MIT Press, 1994.

[HL97a] 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*.

[HL97b] Christoph Armin Herrmann und Christian Lengauer. Transformation of Divide & Conquer to Nested Parallel Loops. In H. Glaser, P. Hartel, und H. Kuchen, Herausgeber, *Programming Languages: Implementation, Logics, and Programs (PLILP'97)*, Lecture Notes in Computer Science 1292, Seiten 95–109. Springer-Verlag, 1997.

[HLG+99] Christoph A. Herrmann, Christian Lengauer, Robert Günz, Jan Laitenberger, und Christian Schaller. A compiler for $\mathcal{HDC}$. Technischer Bericht MIP-9907, Fakultät für Mathematik und Informatik, Universität Passau, Mai 1999.

[HLLS99] Christoph A. Herrmann, Jan Laitenberger, Christian Lengauer, und Christian Schaller. Static Parallelization of Functional Programs: Elimination of Higher-Order Functions & Optimized Inlining. In Iain Duff, Herausgeber, *Euro-Par'99*, LNCS. Springer-Verlag, 1999. To appear.

[HPF97] Paul Hudak, John Peterson, und Joseph Fasel. *A Gentle Introduction to Haskell, Version 1.4*. http://www.haskell.org/tutorial/, März 1997. Online–Dokument.

[Hu99] Zhenjiang Hu. Optimierter Datamining–Algorithmus. Präsentiert beim Dagstuhl Seminar on High-Level Parallel Programming, April 1999.

[JH99a] Simon Peyton Jones und John Hughes. Haskell 98 Report. Technischer Bericht, aktuelle Version bei http://www.haskell.org, 1999.

[JH99b] Simon Peyton Jones und John Hughes. Standard Libraries for Haskell 98. Technischer Bericht, http://www.haskell.org, 1999.

[JHH+93] Simon L Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, und Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*. Keele, DTI/SERC, März 1993.

[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.

[JS98] Simon L. Peyton Jones und André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[KL95] David King und John Launchbury. Structuring Depth-First Search Algorithms in Haskell. In *Proc. 22nd Symposium on Principles of Programming Languages*. San Francisco, Januar 1995.

[KRP92] Owen Kaser, C. R. Ramakrishnan, und Shaunak Pawagi. A New Approach to Inlining. Technischer Bericht, SUNY at Stony Brook, NY, 1992.

[KRP93] Owen Kaser, C. R. Ramakrishnan, und Shaunak Pawagi. On the Conversion of Indirect to Direct Recursion. *ACM Letters on Programming Languages and Systems*, 2(1–4):151–164, März–Dezember 1993.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Nol76] Hartmut Noltemeier. *Graphentheorie*. Walter de Gruyter, 1976.

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

[Sch98] Christian Schaller. *Elimination von Funktionen höherer Ordnung in Haskell-Programmen*. Lehrstuhl für Programmierung. Universität Passau, Fakultät für Mathematik und Informatik, September 1998. Diplomarbeit.

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

[Vol96] Lutz Volkmann. *Fundamente der Graphentheorie*. Springer-Verlag, 1996.

[Wad92] Philip Wadler. The essence of functional programming. In *Proc. 19th Symposium on Principles of Programming Languages*, Seiten 1–14. Albuquerque, New Mexico, Januar 1992.

[YK97] Ting Yu und Owen Kaser. A note on “On the conversion of indirect to direct recursion”. *ACM Transactions on Programming Languages and Systems*, 19(6):1085–1087, November 1997.

Eidesstattliche Erklärung

Hiermit erkläre ich, Jan Laitenberger, eidesstattlich, daß ich die Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 23. Juni 1999