Schleifenparallelisierung in Haskell

Diplomarbeit

Nils Ellmenreich
Universität Passau
18. Dezember 1996

Aufgabensteller und Betreuer:
Prof. Christian Lengauer, PhD.
Lehrstuhl für Programmierung
Fakultät für Mathematik und Informatik
Universität Passau

Danksagung

Besonderer Dank gilt meinem Betreuer und Aufgabensteller Prof. Christian Lengauer, PhD., für viele Diskussionen und Anmerkungen während der Erstellung dieser Arbeit. Weiter seien John T. O'Donnell für seine Erläuterungen zur Sprache Haskell, Michael Mendler für Diskussionen über Monaden, Martin Griebl für die Zusammenarbeit im LooPo-Projekt, Peter Faber für die vorläufige Version seiner SPMD-Schnittstelle im LooPo-Projekt und Christoph Herrmann für das Korrekturlesen gedankt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seit Ende der siebziger Jahre ist das Interesse an funktionalen Sprachen in Forschung und Lehre stetig gestiegen. Schwierigkeiten im Bereich der imperativen Programmierung (*Softwarekrise*) sowie erhöhte Anforderungen in den Bereichen Fehlervermeidung, Wiederverwendbarkeit, Zuverlässigkeit und Geschwindigkeit bei der Erstellung von Software haben zu dieser Entwicklung beigetragen.

Auch auf dem Feld der Parallelprogrammierung wurden funktionale Sprachen interessant. Nachdem in den achtziger Jahren die Verwendung von Parallelrechnern nicht in erwartetem Umfang zugenommen hatte und ihre Programmierung nach wie vor schwierig und die entstandenen Programme schwer portabel waren, versuchten einige Forschungsgruppen, durch Einsatz von funktionalen Sprachen die Maschinenabhängigkeit zu vermindern und somit die eben erwähnten Probleme zu reduzieren. Nach anfänglichen Erfolgen kamen aber auch diese Bemühungen ins Stocken, neue Ansätze fehlten.

Im Bereich der imperativen Programmierung sind Schleifen diejenigen Teile des Programms, in denen ein Hauptteil der Rechenzeit verbraucht wird. Deshalb kommt der Parallelisierung von Schleifen besondere Bedeutung zu. Da die direkte Programmierung von Parallelrechnern, wie schon erwähnt, schwierig ist, haben sogenannte parallelisierende Compiler an Bedeutung gewonnen, die sequentiellen imperativen Code bei der Übersetzung nach potentieller Parallelität untersuchen und dann Zielcode für Parallelrechner erzeugen. Aus diesem Grund ist die automatische Parallelisierung von Schleifen ein wichtiger Bestandteil eines solchen Compilers.

Ein weithin anerkanntes und erfolgreiches Verfahren bei der automatischen Parallelisierung von Schleifensätzen beruht auf dem *Polytopenmodell*, welches aus dem Forschungsgebiet der *systolischen Arrays* stammt. Mit diesem Verfahren wird am Lehrstuhl von Prof. Lengauer intensiv gearbeitet. Im Projekt LooPo ist ein Prototyp eines automatischen Parallelisierers entstanden, in dem verschiedenen Algorithmen, die für das Polytopenmodell entwickelt wurden, implementiert sind. Das Polytopenmodell garantiert unter bestimmten Beschränkungen in der Eingabe eine optimale Parallelisierung.

In dieser Situation ist es naheliegend, daß die Schleifenparallelisierung auf der Basis des Polytopenmodells eine Bereicherung in den Forschungsbemühungen der Parallelisierung funktionaler Sprachen darstellen könnte. Im Rahmen dieser Diplomarbeit soll eine experimentelle Verbindung der derzeit häufig verwendeten funktionalen Programmiersprache Haskell und des Schleifenparallelisierers LooPo entwickelt werden.

Im folgenden wird ein Überblick über den Inhalt der Arbeit gegeben. In Kapitel 2 werden das Polytopenmodell sowie das Projekt LooPo genauer beschrieben. Es folgt in Kapitel 3 eine Darstellung der Programmiersprache Haskell und ihrer Bedeutung innerhalb der Familie der funktionalen Sprachen. Kapitel 4 bringt einen Überblick über die bisherigen Bemühungen in der Parallelisierung funktionaler Sprachen und motiviert die Wahl von Haskell und LooPo als Bindeglieder dieser Arbeit. Zur Darstellung von imperativen Schleifen in Haskell wurde eine Schleifensprache entwickelt, die in Kapitel 5 vorgestellt wird. Um nun Schleifen innerhalb von Haskell parallelisieren zu können, müssen sie in den vorhandenen Konstrukten der Sprache identifiziert werden, da funktionale Sprachen Schleifen im imperativen Sinn (wie z.B. die `FOR`-Schleife) nicht kennen. Welche Sprachelemente von Haskell zur Parallelisierung ausgewählt wurden, wird in Kapitel 6 erläutert. Als nächstes werden Haskell-Ausdrücke mittels eines in Kapitel 7 beschriebenen Übersetzers in Ausdrücke in der Schleifensprache transformiert werden. Diese Schleifensprache kann dann einfach in Schleifensätze der Programmiersprache C umgewandelt werden.

Zu Testzwecken ist es sinnvoll, die Schleifensprache innerhalb von Haskell interpretieren zu können. Eine entsprechende Funktion findet ihre Erläuterung in Kapitel 8. Die Verbindung von rein funktionalen Sprachen wie Haskell und imperativen wie C ist erst in letzter Zeit unter Verwendung der in Kapitel 9 vorgestellten Monaden auf theoretisch fundierte und praktikable Weise ermöglicht worden. Eine unter Benutzung dieser Monaden entwickelte Schnittstelle von Haskell zu C wird in Kapitel 10 dargestellt. Sie ermöglicht es, den gesamten Berechnungsvorgang eines Schleifenäquivalentes in Haskell nach C auszulagern. Die entsprechenden Schnittstellenfunktionen werden automatisch erzeugt und in den Quellcode eingefügt. Ein Beispiel, das alle vorher beschriebenen Konzepte und Funktionen anwendet, ist in Kapitel 11 aufgeführt. Kapitel 12 bringt dann eine Zusammenfassung der Arbeit. Der im Rahmen dieser Diplomarbeit entwickelte Quellcode ist im Anhang aufgeführt.

Kapitel 2

Das LooPo Projekt

Das LooPo-Projekt wurde im Sommer 1994 am Lehrstuhl für Programmierung in Passau mit der Absicht ins Leben gerufen, einige der Arbeiten zur automatischen Parallelisierung im Polytopenmodell zu implementieren, um eine Grundlage zum Sammeln von Erfahrungen mit den eigenen Methoden zu legen. Im Rahmen mehrerer Diplomarbeiten und Praktika wurde unter Leitung von Martin Griebl eine Art Testoberfläche entwickelt, von der aus verschiedene Algorithmen zur Parallelisierung aufgerufen und somit untereinander verglichen werden können.

2.1 Das Polyedermodell

Der Ursprung des Polytopenmodells liegt in den Arbeiten über systolische Arrays, die auf der Arbeit von Karp/Miller/Winograd[KMW67] zu uniformen Rekursionsgleichungen basieren. Eine Übersicht zum Polytopenmodell findet man in [Len93]. Der Ausgangspunkt des Modells ist, die Werte der Laufvariablen von (perfekt) verschachtelten `FOR`-Schleifen in einem entsprechend dimensionierten Koordinatensystem aufzutragen und dadurch jeder Ausführung des Schleifenrumpfes einen Punkt mit ganzzahligen Koordinaten zuzuordnen. Die Menge dieser Punkte bildet das *Quellpolytop*, das aus dem Schnitt von $\mathbb{Z}^n$ und einem konvexen Polytop besteht, welches durch die Schleifengrenzen bestimmt wird. Voraussetzung ist, daß die Schleifengrenzen aus affin-linearen Ausdrücken bestehen, damit es sich überhaupt um ein Polytop handelt.

Innerhalb des Schleifenrumpfes sind im Modell nur Arrayoperationen mit affin-linearen Indexausdrücken zugelassen. Zwischen Arraystatements bestehen Datenabhängigkeiten, wenn auf dasselbe Arrayelement zugegriffen wird. Diese Datenabhängigkeiten werden in den Indexraum als Pfeile zwischen den Punkten eingetragen und ergeben einen gerichteten, azyklischen Graphen. Sie legen eine partielle Ordnung auf den Indexpunkten fest. Die

Einschränkung auf affin-lineare Ausdrücke ist dabei wesentlich, um sicherzustellen, daß die Abhängigkeiten regulär sind.

Im Rahmen der Parallelisierung geht man anschaulich so vor, daß das *Quellpolytop* derart in "Scheiben" zerschnitten (d.h., durch einen Schnitt mit parallelen Hyperbenen partitioniert) wird, so daß zwischen keinen zwei Punkten, die auf einer Hyperbene liegen, Abhängigkeiten existieren und di korrespondierenden Anweisungen daher parallel ausgeführt werden können. Eine solche Unterteilung nennt man *schedule*, und falls jeder Schleifenrumpf frühestmöglich ausgeführt wird, heißt sie *free schedule*. Der *schedule* legt also fest, wann eine Berechnung ausgeführt wird. Aufgrund der affinen Abhängigkeiten liegen alle Punkte, deren Ausführung durch den *schedule* auf den gleichen Zeitpunkt festgelegt wird, auf einer Hyper-Ebene. Zusätzlich muß der Ort der Berechnung, also der Prozessor, festgelegt werden. Während die sogenannte *allocation* eines Polytops der Dimension n ein $n-1$-dimensionaler Unterraum davon ist, hat der *schedule* immer Dimension 1.

Bei der Parallelisierung wird das Quellpolytop durch eine Koordinatentransformation zu dem sogenannten *Zielpolytop* derart verformt, daß die *schedule*-Ebenen senkrecht zur Zeitachse liegen, also von der Zeitachse aufgezählt werden. Die Punkte innerhalb einer *schedule*-Ebene werden entlang der Raumachse nach Vorgabe der *allocation* aufgezählt. Durch die Wahl des *schedules* wird garantiert, daß die Datenabhängigkeiten eingehalten werden und somit keine Berechnung vor oder gleichzeitig mit einer anderen, von der sie abhängig ist, durchgeführt wird. Diese Koordinatentransformation wird auch auf die Indexausdrücke des Quellprogramms angewandt. Die veränderten Ausdrücke werden im Quellprogramm ersetzt, wodurch das parallelisierte Zielprogramm entsteht. Zusätzlich werden die Raumschleifen als parallel gekennzeichnet (z.B. durch Änderung von `for` in `parfor`). Dabei kann gewählt werden, ob die Zeitschleife nach außen gelegt werden soll (synchrone Lösung), oder nach innen (asynchrone Lösung). Im Beispiel im Abbildung 2.1 wurde eine synchrone Transformation gewählt.

Unter Verwendung des *free schedule* wird im parallelisierten Programm jede Berechnung frühestmöglich ausgeführt. Dieser *schedule* kann im Polytopenmodell aber nur dann gefunden werden, wenn er affin-linear ist. In diesem Fall ist die Parallelisierung im Rahmen des Modells optimal.

Diese Übersicht umfaßt nur die einfachste Form des Polytopenmodells. Inzwischen gibt es Erweiterungen, die nicht-perfekt verschachtelte Schleifensätze, `If`-Anweisungen, Prozeduraufrufe und sogar `WHILE`-Schleifen behandeln können. Außerdem dürfen *schedule* und *allocation* von beliebiger Dimensionalität sein, wobei sie für jede Anweisung verschieden und möglicherweise nur stückweise affin sein dürfen [GL96a].

for $i := 0$ to n do
  for $j := 0$ to $i + 2$ do
    $A(i,j) := A(i-1,j)$
      $+A(i,j-1)$
  end
end

for $t := 0$ to $2n+2$ do
  forall $p := \max(0, t-n)$ to $\min(t, \lfloor t/2 \rfloor + 1)$ do
    $A(t-p,p) := A(t-p-1,p)$
      $+A(t-p,p-1)$
  end
end

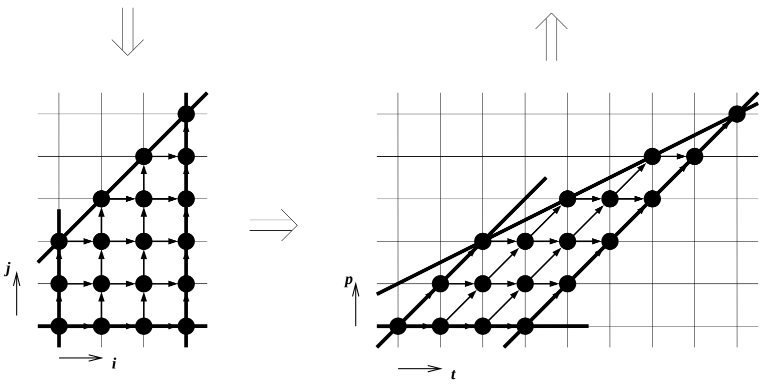

Abbildung 2.1: Parallelisierung im Modell

2.2 LooPo

Wie schon erwähnt, stellt LooPo selbst keinen parallelisierenden Compiler dar, sondern eine Testumgebung, in der Schleifensätze halbautomatisch parallelisiert werden können. Das Projekt ist modular aufgebaut, wobei jedes Modul in der Regel von einem Entwickler betreut wird, um möglichst unabhängiges Arbeiten zu ermöglichen. Ferner können Module vom gleichen Typ (z.B. verschiedene Scheduler) alternativ eingesetzt und somit verglichen werden.

Den Rahmen von LooPo stellen eine Reihe von Hilfsmodulen dar, die in programmiermethodischen Praktika erstellt wurden. Sie umfassen einen Scanner/Parser für Eingabeprogramme [Gün96], die Zielprogrammausgabe [Fab95], ein graphisches Tool zur Anzeige der Indexräume und Datenabhängigkeiten [Wüs95], sowie die graphische Benutzeroberfläche [Ell95].

Den Kern von LooPo stellen Diplomarbeiten dar, die die wesentlichen Module zur Parallelisierung implementiert haben. Sie umfassen die Abhängigkeitsanalyse [Kei96], einen Scheduler/Allocator für Verfahren nach Lamport und Feautrier [Wie95], einen Scheduler/Allocator nach Methoden von Darte, Vivien und Robert [Mei96] und ein Modul zur Zielschleifengenerierung [Wet95]. Weitere Module zur Eliminierung von Datenabhängigkeiten durch *single-assignment conversion*, Partitionierung des Zielraumes zur Anpassung auf reale Architekturen und ein Modul zur Behandlung von `WHILE`-Schleifen sind in Vorbereitung.

Nach Eingabe eines Quellprogramms werden die Hauptmodule in der Reihenfolge Datenabhängigkeitsanalyse, Scheduler, Allocator, Zielschleifengenerierung und Zielprogrammausgabe durch Mausklick aufgerufen. Für jedes Modul existiert ein Optionenfenster, in dem zu verwendende Algorithmen und ihre Parameter ausgewählt werden können. Die Ausgabe der einzelnen Module geschieht in gesonderten Fenstern. Der Benutzer hat so die Möglichkeit, ein Modul mit geänderten Parametern wiederholt aufzurufen, um die Auswirkungen seiner Einstellungen auf die Ausgabe zu erkennen. Die Ansicht des Hauptfensters findet man in Abbildung 2.2.

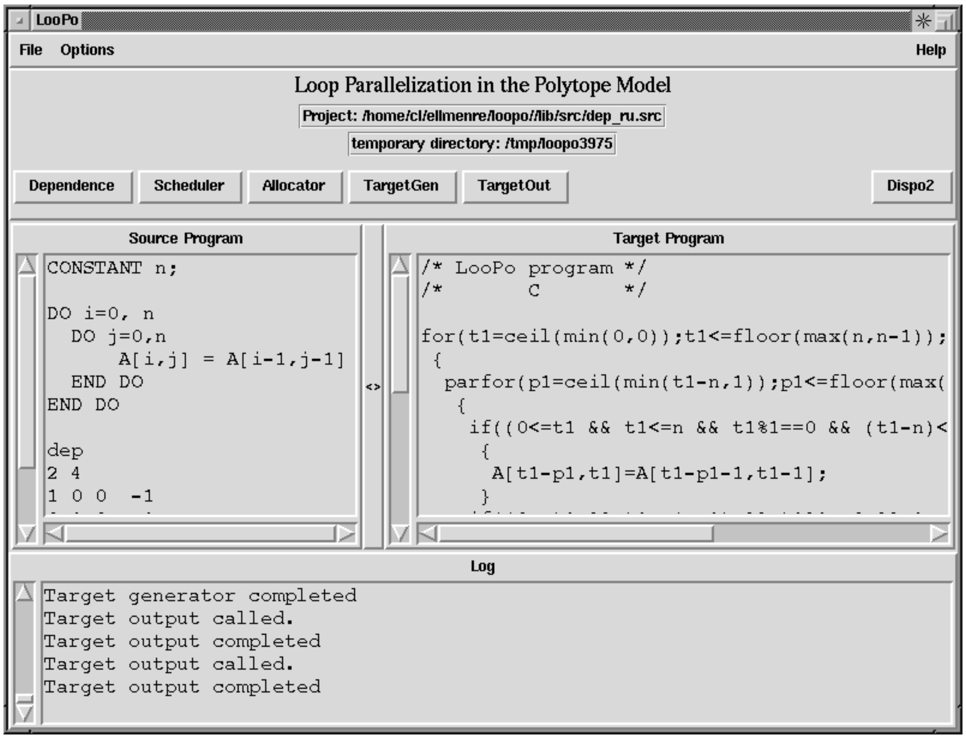

Abbildung 2.2: Das LooPo-Hauptfenster

Auf der linken Seite erkennt man ein Teilfenster, das den zu parallelisierenden Quellcode enthält. Rechts daneben wird nach erfolgreicher Parallelisierung der Zielcode angezeigt. Darüber befindet sich eine Knopfleiste, von der aus die verschiedenen Module von links nach rechts aufgerufen werden. Die Auswahl von verwendeten Algorithmen und Optionen erfolgt über Optionenfenster, die vom zweiten Pulldown-Menü aus erreichbar sind. Über das erste Pulldown-Menü wird mit Hilfe einer Dateiauswahlbox das Quellprogramm geladen und sofort vom Scanner/Parser Modul in die interne Parsebaumdarstellung umgewandelt.

Über dem Zielprogrammfenster befinden sich die Aufrufknöpfe für zwei optionale Module. Dispo ist ein Programm zur graphischen Darstellung der Indexräume und der Datenabhängigkeiten zwischen den Iterationen. Für den Fall, daß die darzustellenden Räume mehr als drei Dimensionen haben oder die Schleifen nicht perfekt verschachtelt sind, können mit dem Filter-Modul Anweisungen und deren Abhängigkeiten herausgefiltert werden, die den Eingabeanforderungen von Dispo entsprechen. Im unteren Teil des Hauptfensters befindet sich ein Log-Fenster, in dem die LooPo Module Meldungen anzeigen können.

LooPo selbst ist frei verfügbar und kann auf verschiedenen UNIX-Plattformen übersetzt werden. Nähere Informationen gibt es auf der WWW-Seite [Loo], im *LooPo User's Guide* [EFG+96] sowie in [GL96a].

2.2.1 Die LooPo-Eingabesprache

LooPo akzeptiert mehrere Eingabesprachen, darunter FORTRAN und C. Allerdings handelt es sich jeweils um Teilmengen der genannten Sprachen, die folgenden Einschränkungen genügen müssen:

- Es werden nur (nicht notwendigerweise perfekt) verschachtelte `FOR`-Schleifensätze mit Schrittweite 1 zugelassen.
- Die Datentypen der Arrays und Skalarvariablen werden von LooPo nicht berücksichtigt, Konstanten müssen vom Typ `Int` oder `Float` sein.
- Die Ausdrücke der Schleifengrenzen müssen affin-linear in umgebenden Schleifenindizes und Strukturparametern sein.
- Als Anweisungen sind nur Arrayzuweisungen erlaubt. Die Arrays dürfen beliebige Dimensionalität haben, ihre Indexausdrücke sind aber höchstens affin-linear. Der Typ der Indizes ist `Int`.
- Der Wertausdruck der Arrayzuweisungen ist ein arithmetischer Ausdruck, der aus den üblichen Grundfunktionen, Variablen und Konstanten zusammengesetzt ist.

Über das Polytopenmodell hinausgehend akzeptiert LooPo in den arithmetischen Ausdrücken auch komplexere Terme als affin-lineare. Diese werden dann aber nicht optimal behandelt; für die Datenabhängigkeiten wird in diesem Fall eine *worst-case* Abschätzung vorgenommen. Erweiterungen für Konditionalanweisungen, Prozeduraufrufe und `WHILE`-Schleifen sind geplant.

Kapitel 3

Die Programmiersprache Haskell

3.1 Funktionale Programmiersprachen

Die Klasse der funktionalen Programmiersprachen bildet zusammen mit den Logiksprachen die Gruppe der *deklarativen Programmiersprachen*, bei denen der Programmtext im wesentlichen beschreibt, *was* zu berechnen ist. Eine schrittweise Berechnungsvorschrift in Form von Anweisungen, wie man sie aus der *imperative Programmierung* kennt, gibt es nicht. Funktionale Programme werden als Funktionen aufgefaßt, die die Eingabe auf die Ausgabe abbilden und aus weiteren Hilfsfunktionen zusammengesetzt sind, während Logikprogramme aus Klauseln bestehen, die aus Fakten Relationen zusammensetzen und Berechnungen in Form von Anfrageklauseln auswerten.

Der Ursprung der funktionalen Sprachen liegt im λ-Kalkül (Church, [Chu41]), der eine Konstruktions- und Auswertungsvorschrift für Funktionen auf ungetypten Daten vorstellt. Später wurden Erweiterungen auf verschiedene Typen eingeführt. Die erste, populäre und noch ungetypte Implementation war *LISP* (*Lis*t *P*rocessing, McCarthy [MAE$^+$64]), später folgten APL und Backus' FP [Bac78].

Man unterscheidet zwischen reinen und nicht-reinen funktionalen Sprachen. Sprachen der zweiten Gruppe, zu denen auch LISP gehört, enthalten zusätzlich zum funktionalen Kalkül einige imperative Sprachkonstrukte, wie zum Beispiel Variablenzuweisungen, Seiteneffekte, Existenz von (globalen) Variablen, *exceptions*, usw. Hierfür gibt es sicherlich einige historische Gründe, aber gerade bei interaktiven Operationen sind imperative Strukturen nur schwer zu umgehen (siehe dazu den Abschnitt 3.2).

Nach Backus' berühmten Artikel *Can Programming be Liberated from the von Neumann Style?* [Bac78], begann eine verstärkte Entwicklung von neuen, insbesondere rein-funktionalen Sprachen. Bekannte Vertreter sind Hope, ML [HMT88] (nicht-rein) und David

Turners Sprachen, von denen in erster Linie Miranda zu nennen ist. Mit der Entwicklung von Graphreduktionsalgorithmen wurden die Implementierungen zunehmend effizienter.

Die Vielzahl der Sprachen behinderte aber den Austausch von Forschungsergebnissen zwischen Gruppen innerhalb der funktionalen Gemeinde; der Ruf nach einer einheitlichen funktionalen Sprache kam auf, die den Stand der Kunst darstellt und als Grundlage der gemeinsamen Forschungstätigkeit dienen sollte. Auf der Konferenz FPCA'87 wurde daher ein Komitee ins Leben gerufen, welches eine solche Sprache entwickeln sollte. Ihr Name ist *Haskell*, benannt nach dem Logiker Haskell B. Curry.

3.2 Funktionale Programmierung

In den letzten Jahren geht die Entwicklung immer mehr hin zu den rein-funktionalen Sprachen. Seit dem einflußreichen, aber nicht rein-funktionalen ML sind im wesentlichen nur noch Sprachen dieser Kategorie entwickelt worden. Sie weisen die im vorigen Abschnitt angesprochenen imperativen Sprachelemente nicht auf und habe daher einige, interessante mathematische Eigenschaften. Im folgenden wird nur noch auf diese Sprachklasse Bezug genommen.

In dem schon angesprochenen Artikel von Backus wird die funktionale Programmierung als Ausweg aus einer Reihe von Problemen gesehen, die in der imperativen Programmierung zur sog. *Softwarekrise* geführt haben. Allgemein anerkannte, positive Merkmale sind:

- Die deklarative Natur der Sprachen: was soll berechnet werden? Maschinenspezifische Details wie die Speicherverwaltung (→ Zeiger), die die Art und Weise, also das "wie" der Berechnung bei imperativen Sprachen stark prägt, sind nicht sichtbar.

- Zerlegung der Lösung eines Problems *top-down* auf natürliche Weise in Funktionen und Unterfunktionen:

  - Homogene Abstraktionshierarchie, d.h. abstrakte Funktionen werden als leere *stubs* zu einem Prototyp zusammengesetzt, bevor sie mit Hilfe von immer konkreteren Hilfsfunktionen ihre eigentliche Funktionalität erreichen (*rapid prototyping*).
  - Einfachere Erstellung und bessere Wartbarkeit der Programme.

- nützliche mathematische Eigenschaften:

  - Die Reihenfolge der Auswertung von Teilausdrücken ist für das Endergebnis unerheblich (→ Church-Rosser Eigenschaft).

– Durch Abwesenheit von Seiteneffekten führt die Anwendung der gleichen Funktion auf die gleichen Parameter immer zu dem gleichen Ergebnis, was die Beweisbarkeit von Eigenschaften des Programms erleichtert (→ *referential transparency*).

Es gibt aber auch einige Nachteile zu nennen:

- Auch Mitte der neunziger Jahre können die meisten Implementierungen von funktionalen Sprachen die Effizienz imperativer Compiler nicht erreichen. Die Gründe hierfür sind die längere Erfahrung im imperativen Compilerbau und die konzeptuelle Nähe imperativer Sprachen zu den heutzutage weit verbreiteten von-Neumann-Architekturen. Allerdings wird der Abstand zunehmend geringer; funktionale Sprachen holen auf.
- Schwierige Umsetzung von stark imperativen Problemstellungen wie Betriebssystemaufgaben und Benutzerinteraktion. Neuere Entwicklungen (→ Monaden, siehe Kapitel 9) erleichtern die Situation für den Programmierer, aber man wird auf diesem Gebiet wohl auf absehbare Zeit Nachteile hinnehmen müssen. Der hohe Abstraktionsgrad der funktionalen Programmierung wird auf Kosten der Effizienz erkauft.

3.3 Haskell

Im folgenden wird ein Überblick über die Sprache Haskell gegeben, bei dem der Schwerpunkt auf den Sprachelementen liegt, die in der vorliegenden Diplomarbeit eine Rolle spielen. Der verwendete Sprachstandard ist Haskell 1.2 [HPJW92], für den es auch eine kurze Einführung gibt [HF92]. Im April 1996 wurde eine langerwartete Überarbeitung verabschiedet (Haskell 1.3 [PHe96]), deren Compiler (ghc-2.01) für die Programmierarbeit zu dieser Diplomarbeit zu spät kam. Als grundlegende Einführungen in die funktionale Programmierung empfehlen sich neben anderen die Bücher von Bird und Wadler [BW87], Thompson [Tho96] und Reade [Rea89].

Haskell ist eine rein-funktionale Programmiersprache mit einer *lazy*-Semantik, d.h. die Auswertung von Funktionsparametern ist erfolgt nach der *call-by-need* Strategie. Das bedeutet, daß die Parameter erst dann ausgewertet werden, wenn sie bei der Berechnung einer Funktion tatsächlich benötigt werden und die Ergebnisse abgespeichert werden, um eine erneute Auswertung zu vermeiden. Unbenötigte Parameter dürfen sogar undefiniert sein, d.h. Haskell ist nicht *strikt*. Funktionen können auch höhere Ordnung haben, d.h. die Rückgabewerte dürfen wieder Funktionen sein. Die Sprache ist stark getypt; als vordefinierte Basistypen gibt es die übliche Menge, u.a. bestehend aus `Int`, `Float`, `Double`, `Char`, `Bool`, usw. Mit `type` kann man Typsynonyme definieren; zusammenfassende Synonyme sind dabei nicht erlaubt:

```
type String = [Char]
type Number = Int | Float   -- verboten !
```

Algebraische Datentypen werden mit einem `data`-Konstrukt erzeugt. Dabei können sowohl die Typ- als auch die Datenkonstruktoren Parameter besitzen:

```
data List a = Nil
            | Cons a (List a)      -- Listendefinition a la LISP
```

Hier ist `List` der Typkonstruktor und `Nil` und `Cons` sind Datenkonstruktoren, d.h. der Ausdruck `Cons 1 Nil` ist vom Typ `List Int`. Das ist auch gleich ein Beispiel für polymorphe Typen in Haskell. Man unterscheidet im allgemeinen zwischen *generellem* und *ad-hoc* Polymorphismus. Letzterer kommt in Haskell in der Form des sogenannten *overloading* in Typklassen vor, bei dem es von einer Funktion mehrere unterschiedliche Implementationen gibt, die nach dem Typ der Parameter ausgewählt werden. Auf Typklassen soll an dieser Stelle nicht weiter eingegangen werden.

Bei universellem Polymorphismus wird bei einer Funktion der gleiche Code für verschiedene Typen ausgeführt (im Gegensatz zum *overloading*, bei dem für verschiedenen Typen eigener Code benutzt wird und nur der Name der Funktion gleich ist). Das läßt sich gut am Beispiel von Listen verdeutlichen, deren Konstruktor in Haskell der Doppelpunkt ist:

```
length:: [a] -> Int
length [] = 0
length (b:bs) = 1 + (length bs)
```

Der Typ einer Funktion wird durch zwei Doppelpunkte nach dem Funktionsnamen angegeben, wobei die einzelnen Parameter durch `->` getrennt werden. Der erste Parameter ist der polymorphe Listentyp, bei dem der Typ der Listenelemente durch die Typvariable `a` gekennzeichnet wird. Die Funktion `length` operiert folglich auf beliebigen Listen. Falls ein Parameter ein algebraischer Datentyp ist, wird, wie im obigen Beispiel, eine Fallunterscheidung über alle Datenkonstruktoren vorgenommen. Rekursive Datentypen und Funktionen sind erlaubt. Wird Typ einer Funktion nicht angegeben werden, so versucht der Typinferenzalgorithmus ihn zur Compilezeit zu bestimmen. Falls das nicht eindeutig möglich ist, bricht die Übersetzung mit einem Fehler ab.

Bei Deklarationen, die sich über mehrere Zeilen erstrecken, wird die Zusammenfassung zu einem Ausdruck durch Einrückung gesteuert. Alternativ kann man auch geschweifte Klammern verwenden und sogar beide Methoden mischen. Kommentare bis zum Zeilenende werden durch `--` gekennzeichnet, Kommentarblöcke beginnen mit `{-` und enden mit `-}`.

Der Konkatenationsoperator für Listen ist `(++)`, einzelne Listenelemente werden in einer Aufzählung durch Kommata getrennt und der LISP-*Cons*-Operator ist in Haskell der

Doppelpunkt. Auf Typen, bei denen ein Nachfolger eindeutig bestimmt ist, können mit `..` Aufzählungen definiert werden. Ein Beispiel:

```
1:[4..7]++[3,5..9] == [1,4,5,6,7,3,5,7,9]
```

Eine weitere Methode zur Konstruktion sind die *list comprehensions* (auch ZF-Notation genannt). Durch eine der mathematischen Mengenschreibweise ähnlichen Notation wird der Ausdruck links des senkrechten Strichs für jede generierte Elementkombination der rechten Seite dupliziert. Zusätzlich sind boolesche Ausdrücke erlaubt, die generierte Elemente filtern können. Beispiele:

```
[ x | x <- [1..5]] == [1,2,3,4,5]
[ x*y | x <- [1..5], odd (x), y <- [3..5]] == [3,4,5,9,12,15,15,20,25]
```

Alle weiteren, üblichen Funktionen höherer Ordnung wie z.B. `fold`, `map` und `scan` existieren auch in Haskell. Eine weitere Illustration ist die sehr elegante Implementation eines einfachen Quicksorts in Haskell:

```
quicksort [] = []
quicksort (a:as) =    quicksort [x | x <- as, x<=a]
                   ++ [a]
                   ++ quicksort [x | x <- as, x>a]
```

Die Ausführung von Funktionsrümpfen kann durch sogenannte *guard patterns* gesteuert werden. Als Beispiel dient hier die Fakultätsfunktion:

```
fac:: Int -> Int
fac n | (n == 0)  = 1
      | (n > 0)   = n * (fac n-1)
      | otherwise = error "Negative argument in function fac."
```

In Abhängigkeit davon, welcher der *guards* wahr wird, ergibt sich der zu berechnende Funktionsrumpf. Der Fall *otherwise* wird gewählt, falls keiner der *guards* wahr ist.

Eine Besonderheit stellen Arrays dar. Der vorgezogene Datentyp in funktionalen Sprachen ist die Liste – für sie gibt es die meisten vordefinierten Funktionen und Algorithmen. Arrays ermöglichen wegen ihrer Entsprechung zum *random access* Speicher einen effizienten Zugriff auf ihre Elemente, was sie zu einem tendenziell imperativen Sprachmittel macht. (Es gibt Leute, die behaupten, Arrays hätten in funktionalen Sprachen nichts zu suchen). Ein Problem stellt die Rezuweisung von Werten an eine Arrayzelle dar, da eine solche Operation die *single assignment* Eigenschaft einer funktionalen Sprache verletzt. Der übliche Ausweg ist, die Zuweisung als eine dreistellige Funktion zu definieren, die als Parameter ein Array, einen Index und einen Wert bekommt und als Ergebnis ein *neues* Array zurückliefert, welches sich vom alten nur an der durch den Index bezeichneten Stelle unterscheidet. Naive Implementationen dieses Schemas sind natürlich extrem ineffizient; tatsächlich führen die meisten Compiler sogenannte *in-place-updates* durch.

Arrays haben in Haskell den Typ `Array a b`, wobei `a` der Typ des Indexes und `b` der Typ der Werte ist. Zur Erzeugung braucht man eine *association list*, eine Liste von Index/Wert Paaren, die die Belegung des Arrays initialisieren. Beispiel:

```
b :: Array (Int,Int) Int
b = array ((0,0),(1,1)) [ ((0,0) := 5), ((0,1) := -4),
                          ((1,0) := 2), ((1,1) := 9)]
```

Hier wird ein zweidimensionales Integer-Array aus vier Elementen definiert. Der Indizierungsoperator ist `!` und der Zuweisungsoperator `//`.

```
b!(1,0) == 2
b//[(0,1):=7] == array ((0,0),(1,1)) [(0,0) := 5,(0,1) := 7,
                                      (1,0) := 2,(1,1) := 9]
```

Zur Konversion einer Liste in ein Array und umgekehrt gibt es zwei Funktionen:

```
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
elems :: Array a b -> [b]
```

Besondere Erwähnung verdient die Funktion `accumArray`, die ähnlich wie `array` eine Initialisierung mit einer *association list* durchführt, nur daß hier in der Liste mehrere Paare mit gleichem Index erlaubt sind. Die Werte dieser Paare werden mit einer assoziativen Operation verknüpft, das Ergebnis wird dann im Array gespeichert.

```
accumArray (+) 0 (0,4) [0:=1, 2:=5, 1:=3, 3:=9, 1:=(-2), 4:=5]
== array (0,4) [0 := 1,1 := 1,2 := 5,3 := 9,4 := 5]
```

Als letzter Aspekt in dieser kurzen Sprachübersicht soll die Ein-/Ausgabe angesprochen werden. Während bis zum Haskell 1.2 Standard *Landin stream-based IO* und *continuation-passing style IO* benutzt wurden, hat man mit der Verabschiedung der Version 1.3 des Standards zu *monadic IO* gewechselt. Der Bedeutung von Monaden in funktionalen Programmiersprachen ist ein eigenes Kapitel (Kapitel 9) gewidmet, so daß an dieser Stelle auf eine Beschreibung verzichtet wird.

Kapitel 4

Vergleich mit paralleler funktionaler Programmierung

4.1 Überblick über die parallele funktionale Programmierung

In vielen Veröffentlichungen der achtziger Jahre wurde die große Bedeutung von funktionalen Sprachen für die Programmierung von Parallelrechnern hervorgehoben. Einer der Hauptgründe hierfür war, daß die Abwesenheit von Seiteneffekten eine gleichzeitige Ausführung von Funktionsargumenten ermöglicht, was bei rein-funktionalen Sprachen zu einem immensen Potential von Parallelität führt. Hinzu kam die Ende der siebziger Jahre geführte Diskussion, die von John Backus an eine breite Öffentlichkeit gebracht wurde [Bac78], wie man den sogenannten *Von-Neumann-Flaschenhals*, also die Verbindung zwischen Prozessor und Hauptspeicher eines Rechners als Engstelle, in zukünftigen Rechnerarchitekturen überwinden könnte. Hier wurden in mehreren Projekten speziell auf funktionale Sprachen zugeschnittenen Rechner entworfen, die sowohl eine Effizienzsteigerung als auch moderneren Programmierstil bringen sollten.

Im folgenden soll ein kurzer Überblick über die funktionale Parallelprogrammierung gegeben werden, um den in dieser Arbeit verfolgten Ansatz in das Gesamtbild einordnen zu können. Einen weiterführenden Überblick findet man auch in [Ham94, Sch93].

4.1.1 (Parallele) Graphenreduktion

Ein Ausdruck in einer funktionalen Sprache wird häufig in einer Baumstruktur dargestellt: der linke Sohn eines Knotens ist die (möglicherweise ge*curry*te) Funktionsdefinition, der rechte Sohn der aktuelle Parameter, und der Knoten selbst repräsentiert die Funktionsanwendung. Im Lambda-Kalkül wird in diesem Baum je nach Auswertungsstrategie einer der reduzierbaren Ausdrücke (*Redex*) ausgewählt und reduziert.

Nachteil dieses Verfahrens ist, daß eventuell mehrfach auftretende Ausdrücke auch mehrfach ausgewertet werden. Um dieses zu verhindern, wird der Auswertungsbaum in einen azyklischen Graphen umgewandelt, in dem gemeinsame Teilausdrücke nur einmal vorkommen und durch Verweise referenzierbar sind. Bei der Auswertung wird somit sichergestellt, daß ein solcher gemeinsamer Ausdruck nur einmal berechnet wird.

Diese zyklischen Graphen werden in der Implementation in eine Menge von Termen umgeformt, die mit Hilfe von erweiterten Termersetzungssystemen ausgewertet werden. Diese *Graphenreduktion* genannte Methode hat sich mittlerweile als Standard bei der Implementierung von funktionalen Sprachen durchgesetzt.

In der parallelen Graphenreduktion wird die Tatsache ausgenutzt, daß sich die verschiedenen Redexe in einem funktionalen Ausdruck gegenseitig nicht beeinflussen können und die Reihenfolge ihrer Auswertung für das Ergebnis irrelevant ist (*Church-Rosser-Eigenschaft* [Pad88]). Folglich kann die Auswertung der Redexe an verschiedene parallele *task*s übergeben werden. An dieser Stelle wird auch schon eine grundlegende Schwierigkeit der funktionalen Parallelprogrammierung deutlich: es gibt zu viele Redexe, als daß es sich lohnen würde, für jeden einen eigenen *task* zu erzeugen, da der zusätzliche Aufwand bei der Erzeugung den eigentlichen Berechnungsaufwand überwiegen kann. Die Frage, wann die Berechnung eines Redexes effizienterweise in einem eigenen *task* durchgeführt wird, ist aber unentscheidbar [PE93].

In den siebziger Jahren waren [Ber75] und [FW76] unter den ersten, die die Bedeutung der unabhängigen Auswertung von Funktionsargumenten sowie die Benutzung von *lazy*-Datenstrukturen in funktionalen Sprachen für die Parallelprogrammierung erkannten.

Die Implementierung hängt entscheidend von der Rechnerarchitektur ab. Bei gemeinsamen Speicher können alle parallelen *task*s auf dem zentralen Programmgraphen operieren, während bei verteiltem Speicher auf lokalen (Teil)Kopien des Graphen gearbeitet wird, was eine zusätzliche Konsistenzsicherung erfordert.

4.1.2 Verschiedene Ansätze

Explizite Parallelität

Der explizite Ansatz existiert sowohl in der imperativen als auch der funktionalen Sprachwelt. Spezielle Anweisungen ermöglichen es dem Programmierer, die Erzeugung parallelen *task*s und deren Plazierung auf Prozessoren direkt zu fordern oder zumindest vorzuschlagen (je nach Programmiersprache). Die verbreitete Familie der *message passing* Sprachen gehört dazu, bei denen auch die Kommunikation zwischen Prozessoren explizit programmiert wird.

Vorteil dieser Technik ist, daß der Programmierer seinen Algorithmus optimal auf die jeweilige Architektur anpassen kann. Allerdings ist dieser Programmierstil in der Regel sehr maschinenabhängig und daher schlecht bis gar nicht portabel. Ein weiterer Nachteil ist ein hoher Programmieraufwand, da diese Methode anfällig für Verklemmungen ist und aufgrund ihres niedrigen Sprachniveaus im nachhinein schwierig zu ändern ist.

In neueren Entwicklungen versucht man zumindest, die Maschinenabhängigkeit zu überwinden, in dem *message passing* Aufrufe als Bibliothek für gängige Architekturen angeboten werden. MPI (*message passing interface)* [DHHW93] ist eine solche Bibliothek, die dem Programmierer eine Schnittstelle für die Programmierung von C und Fortran77 auf verschiedenen Parallelrechnerarchitekturen zur Verfügung stellt. MPI-Programme laufen somit unverändert auf diversen Rechnern, ohne daß die bisher üblichen Anpassungen notwendig sind.

In der funktionalen Programmierung gibt es neben expliziten Anweisungen als Teil der Sprachdefinition auch den Ansatz, Hinweise zur parallelen Auswertung in Kommentaren im Programmtext anzugeben. Diese Kommentare verändern weder die Semantik der Sprache noch die Semantik des Programms, wohl aber die Effizienz der Ausführung. Hudak und Smith haben für diesen Programmierstil in [HS86, Hud86] den Begriff *para-functional programming* geprägt.

Im vorherigen Abschnitt wurde die Schwierigkeit der automatischen effizienten *task*-Erzeugung in der parallelen Graphenreduktion angesprochen. Diese Problematik hat dazu geführt, daß die *task*-Erzeugung häufig durch explizite Sprachelemente gesteuert wird, um diese Entscheidung dem Programmierer zu überlassen. Dieser Ansatz wird von Clean [PE93] favorisiert.

Implizite Parallelität

Der wünschenswerteste Weg zur Parallelisierung von Programmen ist der implizite: der Compiler erkennt zur Übersetzungszeit, ob und welche Teile des Programms unabhängig voneinander sind und daher parallel ausgeführt werden können. Vom Benutzer werden keine Hinweise zur effektiven Durchführung dieser Aufgabe verlangt. Neben der offensichtlich angenehmen Situation, sich beim Programmieren keine Gedanken über eine mögliche parallele Ausführung machen zu müssen, hat dieser Ansatz weitere Vorteile:

- Benutzern mit wenigen oder keinen Kenntnissen über Parallelität wird der Zugang zur Parallelität durch diese Technik überhaupt erst ermöglicht.

- Schon existierende, sequentielle Programme können ohne weiteren Programmieraufwand parallelisiert werden (*automatische Parallelisierung*).
- Durch die Abwesenheit von architekturspezifischen Anweisungen ist der Code portabel und einfacher zu programmieren. Die Auseinandersetzung des Programmierers mit der Abbildung des Algorithmus auf einen Parallelrechner mittels expliziter Sprachkonstrukte bedarf entsprechender Kenntnisse und ist so aufwendig, daß dieser Umstand als ein Hauptgrund für die schleppende Durchsetzung der Parallelprogrammierung angesehen werden kann.

Allerdings ist in der Literatur mehrfach argumentiert worden, daß die automatische Erkennung von Parallelität nur in Spezialfällen effizient sein kann und für einen allgemeinen Ansatz nur der explizite in Frage komme. Dieses Argument betrifft aber hauptsächlich imperative Sprachen, da für funktionale genau das Gegenteil richtig ist: Funktionale Teilausdrücke sind wegen fehlender Seiteneffekte voneinander unabhängig und sogar die Reihenfolge ihrer Auswertung hat keine Auswirkung auf ihr Ergebnis. Somit steht zumindest theoretisch ein massives Potential an Parallelität zur Verfügung. Das Problem ist aber der Aufwand für die Erzeugung eines weiteren parallelen *task*s zur Berechnung eines Ausdrucks. Er ist in der Regel so groß, daß er sich für kleinere Ausdrücke nicht lohnt. Die automatische Entscheidung, wann es sich lohnt, einen Teilausdruck durch einen neuen parallelen *task* auswerten zu lassen, ist sehr schwierig. Diese Problematik ist der Hauptgrund, weshalb bisherige Ansätze überwiegend expliziter Natur sind.

Funktionen höherer Ordnung

Die folgenden Ansätze sind nicht klar in die oben beschriebenen Kategorien einzuteilen. Funktionen höherer Ordnung werden benutzt, um auf einer abstrakten Ebene Operationen zu definieren, deren Implementierung parallel ausgeführt werden kann. Da in der jeweiligen Sprache keine expliziten Anweisungen zur Steuerung der parallelen Bearbeitung vorhanden sind, kann man von einem eher impliziten Ansatz sprechen. Auf der anderen Seite sind in einer gewählten Sprachumgebung die Implementierung der Funktionen höherer Ordnung auf dem Parallelrechner häufig fest gewählt, so daß das parallele Verhalten allein durch Auswahl der Funktionen direkt beeinflußt werden kann. Aus diesem Grund schätzen manche Leute diesen Ansatz als eher explizit ein.

Bei der Auswertung bestimmter Funktionen wie z.B. `map` und `fold` ([PHe96]) werden gleichartige Operationen auf allen Elementen einer Liste ausgeführt. Datenparallelität versucht, möglichst viele dieser Operationen parallel auszuführen, weshalb sich dieses Schema insbesondere für SIMD Architekturen eignet.

Listen sind aber nicht der einzige Datentyp für diese Methode; Hill [Hil92] entwickelt in der datenparallelen Haskell-Version *DPHaskell* den Datentyp `pod` (*parallel object with* *arbitrary dimension*), der eine Menge von potentiell unendlich vielen Index/Wert-Paaren beinhaltet. Eine *comprehension*-Syntax ähnlich der von Haskell ermöglicht datenparallele Berechnungen auf diesem Typ.

Im Bird-Meertens Formalismus (*BMF*) [Ski94] werden auf einer abstrakten Ebene Funktionen höherer Ordnung zur methodischen Programmentwicklung benutzt. Dabei existieren Implementationen dieser Funktionen für Parallelrechner. Geschachtelte Ausdrücke können mit Hilfe von Transformationsregeln semantisch äquivalent bei gleichzeitiger Reduzierung der Kosten (bzgl. eine Kostenkalküls) umgeformt werden.

Der von Murray Cole geprägte Begriff *algorithmic skeleton* [Col89] bezeichnet die Verwendung polymorpher Funktionen höherer Ordnung, für die jeweils für eine Rechnerarchitektur eine konkrete Parallelisierung existiert. Durch den Typ der Funktion wird garantiert, daß jeder Aufruf dem Schema entspricht und daher nach der für dieses Skelett existierenden Methode parallelisiert werden kann. Skelette werden häufig bei Divide-and-Conquer Parallelisierungen benutzt, da sich viele Probleme aus diesem Bereich auf einige Schemata und somit Skeletons reduzieren lassen.

4.2 Warum Haskell und LooPo verbinden?

Wie im vorigen Abschnitt bereits beschrieben wurde, lassen sich die Ansätze zur Einführung von Parallelität in Programmiersprachen im wesentlichen in den impliziten und den expliziten Zugang einteilen. Die automatische Parallelisierung von sequentiellen Programmen kann sich natürlicherweise auf keine expliziten Konstrukte stützen – daher ist das Ziel des in Kapitel 2 beschriebenen Polytopenmodells die implizite Parallelisierung.

Die Mehrzahl der bisherigen Ansätze zur Parallelisierung funktionaler Sprachen ist allerdings explizit; der Grund dürfte in der in den vorigen Abschnitten beschriebenen Schwierigkeit liegen, im impliziten Fall die massive feinkörnige Parallelität effizient in den Griff zu bekommen. Selbst Projekte wie GUM [THMJ+96], die als Ziel die automatische Parallelität ohne spezielle Direktiven haben, beschränken sich zur Zeit auf die Benutzung von expliziten `seq` und `par` Konstrukten, um Erfahrungen mit ihrem Ansatz zu machen.

Allgemein gibt es im impliziten Fall zwei mögliche Ansätze:

- Parallelisierung des gesamten Programmes nach erfolgter Datenabhängigkeitsanalyse. Hier können Modelle wie z.B. Petrinetze oder Prozeßkalküle zum Einsatz kommen. Die Erfahrung zeigt aber, daß der erreichte Parallelitätsgrad eher enttäuschend ist, da in der Regel keine speziellen Methoden angewendet werden, die die besonderen Eigenschaften bestimmter Programmkonstrukte wie zum Beispiel Schleifen bezüglich ihrer möglichen Parallelität ausnutzen. Somit ist das Ergebnis von eher zu grobkörniger Natur.

- Parallelisierung von Teilen des Programms (wie z.B. Schleifen), für die spezifische Methoden existieren. Der parallelisierte Teil wird dann in das (möglicherweise noch sequentielle) Hauptprogramm eingebettet. Der Vorteil besteht in einer besseren, zum Teil sogar optimalen Ausnutzung der lokalen Eigenschaften eines Konstruktes.

Es ist offensichtlich einfacher und ergibt somit effizientere Ergebnisse, wenn man, wie im zweiten Fall, nur begrenzte Teile eines funktionalen Programms betrachtet und für diese dann eine möglichst optimale Parallelisierung findet. Dieser Weg soll im folgenden gegangen werden, um das Polytopenmodell anwenden zu können.

In den folgenden Kapiteln soll eine mögliche Verbindung einer funktionalen Sprache mit dem Polytopenmodell untersucht werden, die für die oben beschriebenen Probleme eine Alternative darstellt. Als Vertreter der beiden "Welten" wurden LooPo [GL96b] und Haskell [HPJW92] gewählt.

Der Transfer dieses aus der imperativen Welt stammenden Konzeptes der Schleife erfolgt unter der Annahme, daß auch in funktionalen Programmen schleifenähnliche Programmsegmente signifikant an der Gesamtlaufzeit beteiligt sind. Im weiteren könnte dann die Schleifenparallelisierung zusammen mit anderen "Spezialmethoden" zu einem Gesamtkonzept zusammengefügt werden.

Kapitel 5

Die imperative Schleifensprache

5.1 Warum eine imperative Schleifensprache?

Im Kapitel 2 wurde das Polytopenmodell als Methode zur automatischen Parallelisierung von (imperativen) Schleifensätzen mit Arrayzugriffen beschrieben. Um nun dieses Verfahren im Rahmen von funktionalen Sprachen einsetzen zu können, muß eine Schleifensprache innerhalb der funktionalen Sprache entwickelt werden, deren Programme einfach in eine Eingabe für einen automatischen Parallelisierer wie z.B. LooPo umgewandelt werden. Dahinter steht die Absicht, bestimmte funktionale Ausdrücke in Schleifen umzuwandeln, um sie dann parallelisieren zu können.

Diese Voraussetzungen stellen folgende Anforderungen an die Schleifensprache:

- Sie muß in Haskell ausdrückbar sein.
- Sie sollte einfach in Haskell interpretierbar sein.
- Sie sollte mindestens die Mächtigkeit der Eingabesprache von LooPo haben oder — besser noch — der vollen Allgemeinheit des Polytopenmodells genügen und einfach erweiterbar sein.
- Es müssen höherdimensionale Arrays mehrerer numerischer Datentypen existieren.

Unsere implementierte Schleifensprache erfüllt diese Anforderungen. Sie enthält die Datentypen `DOUBLE` und `INT`, Variablenzuweisung, mehrdimensionale Arrays, das `IF`-Konstrukt, `FOR`- und `WHILE`-Schleifen sowie arithmetische Ausdrücke. Sie kann in einer Haskell-Laufzeitumgebung interpretiert werden oder auch nur einen entsprechenden C-Schleifensatz umgewandelt werden.

5.2 Algebraische Datentypen für die Schleifensprache

Die im folgenden beschriebenen Typen sind Teil des im Rahmen dieser Diplomarbeit implementierten Programms, welches die Haskell-Schnittstelle zu LooPo umfaßt. Zur Repräsentation der Sprache wurden algebraische Datentypen gewählt, da eine solche Repräsentation innerhalb von Haskell sehr natürlich ist. Es wird jeweils eine Datentyp für die arithmetischen Ausdrücke und die Anweisungen der Schleifensprache definiert.

5.2.1 Arithmetische Ausdrücke

Die zugehörige `data` Anweisung führt einen neuen Typ namens `Exp` ein. Jeder Ausdruck dieses Typs muß einen Typkonstruktor beinhalten, der in der nachstehenden Typdeklaration definiert ist.

```
data Exp =
    IConst Int
  | DConst Double
  | Contents Var
  | ArrContents Var Index
  | Exp :+: Exp
  | Exp :@: Exp
  | Exp :>: Exp
  | Exp :<=: Exp
  | Exp :*: Exp deriving (Text)
```

Der Typ umfaßt zum einen die Präfixkonstruktoren `IConst` und `DConst`, die die entsprechenden Haskell-Typen in `Exp` einbetten, sowie `Contents` und `ArrContents`, die den Wert der angegebenen Skalarvariablen bzw. des Arrays an einer Stelle darstellen. Dabei ist `Var` ein Typsynonym für `String` und stellt den Bezeichner einer Variable dar; `Index` ist ein eigener Datentyp, der mehrdimensionale Indexausdrücke umfaßt (s.u.).

Da algebraische Datentypen zwingend Typkonstruktoren vorschreiben, mußten die Typen `Int` und `Double` mit jeweils einem Label versehen werden, um sie in den Typ `Exp` einzubetten. Diese Einbettung kann zu Problemen führen, wenn der eigentliche Basistyp wieder extrahiert werden soll. Folgende Funktionsdefinition ist in Sprachen mit dem Hindley-Milner Typsystem[DM82] leider nicht möglich:

```
getNum:: Exp -> a
getNum (IConst n) = n
getNum (DConst f) = f
```

Nach Auswertung der zweiten Zeile wird der Typ `a` vom Typinferenzalgorithmus zu `Int` spezialisiert, woraufhin der Compiler in der dritten Zeile einen statischen Typfehler meldet, da dieser Ausdruck von Typ `Double` und nicht `Int` ist. Aus diesem Grund wird der Rückgabewert der `accumarray` Funktion (siehe Kapitel 8) auch ein Aggregattyp und kein Array sein. Bei den derzeitigen implementierten Inferenzalgorithmen gibt es keine andere Möglichkeit.

Zum anderen existieren Infixoperatoren, die jeweils zwei Ausdrücke mittels einer arithmetische Operation verknüpfen. Hier existieren die Addition, Subtraktion[1] und Multiplikation sowie die Größer-Gleich- und Kleiner-Gleich-Relation.

Durch die Verwendung eines algebraischen Datentyps wird insbesondere mit den Infixoperatoren eine natürliche Darstellung innerhalb von Haskell erreicht. Die Summe zweier Integer-Zahlen wird z.B. durch `((IConst 5) :+: (IConst 3))` dargestellt.

Durch Erweiterung des Datentyps `Exp` können einfach neue Operationen und Typen eingeführt werden. Allerdings müssen dann Funktionen wie der Interpreter und Übersetzer, die `Exp` verarbeiten, auch entsprechend erweitert werden.

Schließlich wird noch der Typ `Index` verwendet, der verschiedendimensionale Indizes zusammenfaßt. Auch hier bereitet eine Einschränkung der derzeitigen Sprachdefinition von Haskell Probleme. Die Indizes von Arrays in Haskell werden mit Hilfe von Tupeln angegeben – für die interne Schleifensprache wäre daher ein Typ "n-Tupel von `Exp`" wünschenswert, der dann später bei der Indizierung eines Arrays in ein "n-Tupel von `Int`" umgewandelt werden könnte. Solche n-Tupel Typen sind in Haskell aber nicht möglich; die hier beschriebene Schleifensprache benutzt daher einen algebraischen Datentyp, der einige Dimensionen explizit aufzählt. Somit ist eine allgemeine Behandlung von n-dimensionalen Arrays nicht möglich, wohl aber die Verwendung von Arrays bis zu einer beliebigen, fest gewählten Dimensionalität.

Als Beispiel sei der Typ `Index` für ein- und zweidimensionale Indizes gewählt:

```
data Index = Ix1 Exp
           | Ix2 Exp Exp deriving (Text)
```

Hier bekommt also wieder jede Dimension einen eigenen Typkonstruktor, der zu entsprechend vielen Fallunterscheidungen in betroffenen Funktionen führt. Als Beispiel sei der Wert des Arrays `A` an der Stelle `(2,1)` mit 4 multipliziert gegeben:

```
((IConst 4) :*: (ArrContents "A" (Ix2 (IConst 2) (IConst 1)))).
```

Bei der Betrachtung dieser Beispiele darf man nicht vergessen, daß es sich bei dieser Sprache immer noch um eine interne Darstellung handelt, die nicht zur direkten Programmierung

[1]Der Operator heißt hier leider `:@:`, da Haskell 1.2 das naheliegendere `:-:` syntaktisch nicht zuläßt

durch einen Benutzer gedacht ist. Für einen solchen Zweck bieten die algebraischen Datentypen eine gut lesbare Darstellung.

5.2.2 Die imperativen Anweisungen

Da die Typen `Exp` und `Index` schon definiert sind, läßt sich jetzt die eigentliche imperative Minisprache, repräsentiert durch den Typ `Cmd`, einfach definieren. Neben der leeren Operation `Nop` existiert die Befehlssequenz als Infixoperation (`:>>`), ein `IF`-Konstrukt, die `FOR`- und `WHILE`-Schleife, sowie Zuweisungen an Skalar- und Arrayvariablen. Eine Sonderrolle nehmen `DefArray` und `InitArray` ein; während das erste nur ein Array von angegebener Größe, aufgefüllt mit einem Nullwert, anlegt, wird durch das zweite zusätzlich eine Vorbelegung definiert. Der resultierende Datentyp sieht wie folgt aus:

```
data Cmd =
    Nop
  | Cmd :>> Cmd
  | If Exp Cmd Cmd
  | While Exp Cmd
  | For Var Exp Exp Cmd
  | Assign Var Exp
  | ArrAssign Var Index Exp
  | DefArray Var Index Val
  | InitArray Var Index [Val]
```

Der einzige bisher noch nicht definierte Datentyp ist `Val`. Er faßt als Aggregattyp diejenigen Typen zusammen, die als Elemente von Arrays existieren können. Seine Existenz hat den gleichen Grund wie die Einbettung von `Int` und `Double` in `Exp`: die Beschränkungen, die durch die Typinferenzalgorithmen auferlegt werden. Die Definition sieht daher wir folgt aus:

```
data Val = I Int
         | D Double deriving (Eq, Ord, Text)
```

Damit können imperative Schleifenprogramme in Haskell als Ausdrücke vom Typ `Cmd` behandelt werden. Ein Programm, das die Zweierpotenzen bis 2^{10} in einem Array abspeichert, hat dann folgende Form:

```
(DefArray "A" (Ix1 (IConst 10)) (I 0)) :>>
(Assign "N" (IConst 1))                :>>
(For "I" (IConst 0) (IConst 10)
    ((ArrAssign "A" (Ix1 (Contents "I")) (Contents "N")) :>>
     (Assign "N" ((Contents "N") :*: (IConst 2))))
)
```

Die Schleifensprache ist mächtiger bzw. allgemeiner, als für die in dieser Diplomarbeit benutzten Schleifenprogramme erforderlich. Es erschien aber sinnvoll, die für eine mögliche Erweiterung der Arbeit notwendigen Grundlagen jetzt schon zu legen und damit zu illustrieren, daß in diesem Fall durch die implementierte Schleifensprache keine prinzipiellen Beschränkungen bestehen.

Kapitel 6

Wie wird LooPo mit Haskell verbunden?

6.1 Welche Haskell-Ausdrücke sollen parallelisiert werden?

Um das Polytopenmodell in einer funktionalen Sprache erfolgreich einsetzen zu können, müssen zuerst diejenigen Sprachkonstrukte identifiziert werden, die ihrer Natur nach einer Schleife entsprechen. Diese können dann zumindest prinzipiell parallelisiert werden.

Der hier eingeschlagene Weg, den imperativen Parallelisierer LooPo zu verwenden, beschränkt die tatsächlich bearbeitbaren funktionalen Konstrukte aufgrund seiner restriktiven Eingabesprache. Im folgenden werden potentielle Kandidaten für eine Schleifenparallelisierung vorgestellt, von denen sich leider einige, wie sich zeigen wird, für die Bearbeitung in LooPo nicht eignen.

Eine naheliegende Entsprechung von Schleifen in funktionalen Sprachen stellen rekursive Funktionsaufrufe da. Obwohl insbesondere im Fall der Endrekursion die Umsetzung in eine `FOR`-Schleife trivial ist, sind hier durch die Möglichkeit der Anwesenheit beliebiger funktionaler Ausdrücke im Schleifenrumpf Grenzen für den sofortigen Einsatz von LooPo gegeben. Als Spezialfälle von Endrekursionen existieren eine Reihe von Funktionen höherer Ordnung, welche auf Listen operieren (`map`, `fold`, `scan`, usw.) Eine Parallelisierung dieser verbreiteten Funktionen würde für viele funktionale Programme einen erheblichen Zuwachs an Parallelität bedeuten. Gerade diese Funktionen stellen bei der Transformation in ein Schleifenprogramm ein Problem dar. Wie in Kapitel 2 beschrieben wurde, arbeitet die bisherigen Verfahren zur Datenabhängigkeitsanalyse im Polytopenmodell auf einer Eingabesprache aus Schleifensätzen mit Arrayoperationen und Funktionen erster Ordnung. Diese Algorithmen sind für `fold` und ähnliche Funktionen, die als Argumente wieder beliebig komplexe funktionale Ausdrücken haben können, nicht ausgelegt. Hier wäre ein erheblicher Anpassungsaufwand erforderlich, der im Rahmen dieser Arbeit nicht geleistet werden kann, möglicherweise aber in der Zukunft ein erfolgversprechender Weg ist.

Es ist auch denkbar, das Polytopenmodell in einen parallelisierenden Compiler einer funktionalen Sprache direkt einzubauen, ohne den Umweg über LooPo und eine imperative Sprache zu gehen. Hier könnten dann weiterentwickelte Methoden der Abhängigkeitsanalyse direkt eingesetzt werden.

In allen bisher beschriebenen Alternativen ist es zwar denkbar, nur von LooPo verarbeitbare Ausdrücke zuzulassen. Dies erscheint aber nicht sinnvoll, da die Einschränkungen zuviel von der Ausdruckskraft dieser Funktionen eliminieren und nur künstlich wirkende Konstruktionen übrig lassen.

Es muß also eine Menge von Ausdrücken in Haskell gefunden werden, die mit vernünftigem Aufwand für LooPo übersetzbar ist, bei denen eine Einschränkung auf arithmetische Ausdrücke die Natur der ausgewählten Funktionen nicht zu sehr verändert und deren Komplexität groß genug ist, um überhaupt eine Parallelisierung sinnvoll erscheinen zu lassen.

Unter diesen Gesichtspunkten kommt man zwangsläufig auf die Haskell eigenen Arrays, aus denen sich Ausdrücke zusammensetzen lassen, deren Transformation in einen Schleifensatz den Anforderungen von LooPo entspricht. Arrays in rein-funktionalen Sprachen haben die *single assignment* Eigenschaft, die nach der Initialisierung erneute Zuweisungen verbietet. Um trotzdem die aus imperativen Sprachen gewohnten Arrayoperationen nutzen zu können, werden Operationen, die den Inhalt eines Arrays verändern, als Funktionen gesehen, die als Parameter ein Array sowie eine Liste von Index/Werte-Paaren erhalten und ein neues Array zurückliefern. Dieses neue Array unterscheidet sich vom alten durch die neuen Werte an den durch die Indizes angegebenen Stellen. Die Tatsache, daß einige Haskell-Compiler dieses ineffiziente Vorgehen durch sogenannte *in-place updates* ersetzen, die kein neues Array anlegen, ist ein reines Implementationsdetail.

Somit gilt es also zu untersuchen, inwieweit sich Konstruktionsfunktionen von Arrays für unsere Zwecke eignen.

listArray ist die einfachste und expliziteste Art, ein Array zu erzeugen – es wird einfach eine Liste der Elemente angegeben, mit denen das Array in der angegebenen Reihenfolge aufgefüllt wird. In der Regel werden die Werte explizit angegeben, so daß gar keine parallelisierbaren Berechnungen stattfinden. Sollte die Liste Ausdrücke enthalten, die erst noch ausgewertet werden müssen, so können diese Berechnungen trivialerweise alle parallel ausgeführt werden, da in der Liste auf keine der neuen Arrayelemente zugegriffen werden darf und somit keine Datenabhängigkeiten bestehen können.

Ein besondere Form der Listendarstellung ist die *list comprehension* (auch ZF-Notation genannt), die der mathematischen Mengenschreibweise ähnelt (siehe Kapitel 3). Wenn eine solche Darstellung keine booleschen Filter enthält, entspricht sie direkt einem Schleifensatz, bei dem allerdings mit der gleichen Begründung wie oben alle Iterationen unabhängig voneinander sind. Falls der Generatorsausdruck den LooPo-Eingabebeschränkungen folgt, ist der gesamte Schleifensatz eine legale LooPo-Eingabe.

Mit einfachen arithmetischen Ausdrücken oder sogar Konstanten ist die parallelisierte Berechnung der Arrayelemente daher trivial, bei Verwendung von Funktionen höherer Ordnung gibt es aber die eingangs erwähnten Probleme.

array ist zu `listArray` äquivalent, nur daß sich die Konstruktion nicht auf einer Werteliste begründet, sondern einer sogenannten *association list* (siehe Kapitel 3); in diesem Fall spricht man von einer *array comprehension*. Für die Parallelisierung gilt das gleiche wie im vorherigen Fall.

accumArray ist eine Erweiterung von `array`, bei der in der *association list* mehrfach Index/Wert-Paare mit gleichem Index auftreten können. Die Werte dieser Paare werden zuerst mit einer assoziativen, binären Operation verknüpft, bevor das Ergebnis in die entsprechende Arrayzelle eingetragen wird. Auch diese Funktion wird typischerweise mit einer *array comprehension* aufgerufen.

Die Berechnungen für jeden Wert einer Arrayzelle sind zwar wieder voneinander unabhängig, aber da mehrere Index/Wert-Paare für den gleichen Index vorkommen können, sind die Iterationen des aus der *comprehension* resultierenden Schleifensatzes nicht unbedingt unabhängig.

Durch die Akkumulation von Iterationsergebnissen können die oben genannten Algorithmen Polynomprodukt und Matrixmultiplikation in natürlicher Weise ausgedrückt werden.

Hier besteht also ein Sprachkonstrukt, welches sich in einen Schleifensatz übersetzen läßt, der (unter den bei `listArray` angegebenen Einschränkungen) eine legale LooPo-Eingabe darstellt. Das Konstrukt ist mächtig genug, um interessante Algorithmen darzustellen, und die Frage der Datenabhängigkeiten ist nicht auf triviale Weise zu beantworten. Für eine effiziente Bearbeitung durch das Polytopenmodell dürfen nur affine Datenabhängigkeiten zwischen den Arrayelementen bestehen. Da LooPo aber auch komplexere Abhängigkeiten verarbeiten kann (wenn auch nicht optimal), wird hier auf eine entsprechende Beschränkung der Eingabe verzichtet.

Aus den soeben dargestellten Überlegungen ergibt sich, daß im Rahmen dieser Diplomarbeit `accumArray`-Ausdrücke untersucht werden sollen, deren Wertausdruck ein einfacher arithmetischer Ausdruck ist und dessen *list comprehension* keine booleschen Filter enthält.

Ein solcher Ausdruck hat allgemein die folgende Struktur:

```
accumArray ::
  (Ix a) => (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b
```

Die *array comprehension* der *association list* hat folgende Form:

$$\texttt{[(}\mathit{index\_expr}\texttt{:=}\mathit{val\_expr}\texttt{) |}\mathit{qual}_1, \ldots, \mathit{qual}_n \texttt{]}$$

wobei ein *qualifier* entweder ein Indexgenerator oder ein boolescher Filter ist.

Der zugehörige, imperative Schleifensatz hat dann folgende Struktur:

for ... $qual_1$ ...
  $\ddots$
    for ... $qual_n$ ...
      `_result`[*index_expr*] := `_result`[*index_expr*] '*assoz.op*' *val_expr*

Während dem Ergebnisarray in Haskell nicht notwendigerweise ein Name gegeben werden muß, ist das aufgrund der Einzeloperationen im Imperativen notwendig. Standardmäßig wird hier vom implementierten Übersetzer in die Schleifensprache (siehe Kapitel 7) der Name `_result` vergeben. An dieser Stelle ist zu erwähnen, daß im Rahmen des Polytopenmodells die Assoziativität des Verknüpfungsoperators nicht ausgenutzt wird. Das könnte zu logarithmischen Schleifengrenzen führen, mit denen das Modell nicht umgehen kann.

6.2 Die Parallelisierung von `accumArray`

Wie im vorigen Abschnitt dargelegt wurde, soll die Haskell-Funktion `accumArray` geeignet ersetzt werden, so daß ihre Transformation in einen imperativen Schleifensatz nach der Polytopenmethode parallelisiert werden kann. Dazu beginnt man mit einem normalen Haskell-Programm, dessen `accumArray` Aufrufe folgenden Einschränkungen genügen:

- Die der `accumArray`-Funktion übergebene Liste muß in der *array comprehension* Form dargestellt werden, da sonst keine nutzbare Quelle für Parallelität existiert. Ferner dürfen die *qualifier* der *comprehension* keine booleschen Filter enthalten; diese würden in eine `if`-Anweisung übersetzt werden müssen, welche LooPo nicht verarbeiten kann.

- Der Wertausdruck der *comprehension* darf nur ein rein arithmetischer Ausdruck sein, d.h. er darf insbesondere keine Funktionen höherer Ordnung enthalten, die nicht in die LooPo-Eingabesprache übersetzt werden können. Für optimale Ergebnisse der Parallelisierung im Polytopenmodell sollte der Anwender sich auf affine Ausdrücke beschränken. Nicht-affine Teilausdrücke, wie z.B. ein quadratischer Term, können von derzeitigen Algorithmen zur Abhängigkeitsanalyse nicht direkt bearbeitet werden; als Ausweg werden von LooPo maximale Abhängigkeiten angenommen, d.h. im betreffenden Teilausdruck ist jeder Index von jedem anderen abhängig, was die Berechnung zwangsläufig sequentialisiert.
- Um den automatischen Codegenerator nicht unnötig zu komplizieren, müssen die Parameter für `accumArray` in der gleichen Zeile hintereinander aufgeführt sein.

Zum Austesten seines Programms bewegt sich der Benutzer vollständig in der sequentiellen Welt und kann sogar einen Haskell-Interpreter wie Hugs [Jon96] zur Entwicklung benutzen. Sobald er fertig ist und sich bei der Benutzung von `accumArray` an die oben aufgeführten Einschränkungen gehalten hat, kann er mit der Parallelisierung beginnen. Die Idee dabei ist, die Berechnung von `accumArray` auf einem Parallelrechner durchzuführen, dessen Ansteuerung in der Regel durch Bibliotheken in der Programmiersprache C geschieht. Folglich muß eine Schnittstelle zwischen Haskell und C geschaffen werden, die es ermöglicht, anstelle von `accumArray` eine C-Funktion aufzurufen, der als Parameter alle zur Berechnung benötigten Variablen übergeben werden. Diese C-Funktion soll wiederum die Berechnung des durch `accumArray` implizit definierten Schleifensatzes auf einem Parallelrechner starten und das Ergebnis an das Haskell-Programm zurückliefern.

Die Parallelisierung des sequentiellen Schleifensatzes erfolgt durch LooPo. Da zum Zeitpunkt der Fertigstellung dieser Diplomarbeit die LooPo-Schnittstelle zu einem Parallelrechner noch nicht fertig implementiert ist, beschränke ich mich darauf, die Schnittstelle von Haskell zu C automatisch zu generieren und innerhalb der C-Funktion die Berechnung durch den sequentiellen Schleifensatz durchführen zu lassen.

Der automatische Codegenerator, der als Eingabe das Haskell-Programm erhält, führt folgende Operationen durch:

- Die `accumArray`-Aufrufe werden im Haskell-Quellcode identifiziert.
- Unter Verwendung eines in Kapitel 7 beschriebenen Parsers werden die Parameter von `accumArray` in die in Kapitel 5 beschriebene Schleifensprache übersetzt.
- Die `accumArray` Ausdrücke im ursprünglichen Quellcode werden durch Funktionsaufrufe ersetzt. Die entsprechenden Haskell-Funktionen werden automatisch generiert und enthalten den Aufruf einer C-Funktion, die alle zur Berechnung notwendigen Werte als Parameter erhält und das Ergebnis-Array zurückliefert.

- Eine C-Funktion wird automatisch erzeugt, die den oben beschriebenen (hier sequentiellen) Schleifensatz enthält, der vom in Kapitel 7 beschriebenen Übersetzer generiert wurde. Sie führt unter Verwendung der übergebenen Werte die Berechnung des Ergebnis-Arrays durch und liefert dieses zurück.

Der für den Benutzer übliche Ablauf ist damit die sequentielle Entwicklung seines Programms, gefolgt von einer Geschwindigkeitsoptimierung durch nachfolgende automatische Parallelisierung.

Im Rahmen dieser Diplomarbeit wurde der oben beschriebene automatische Codegenerator als Prototyp implementiert, um Erfahrungen für weitergehende Anstrengungen in der automatischen Parallelisierung von funktionalen Programmen zu sammeln. Da auch die Weiterentwicklung der Schleifensprache möglich sein soll, ist es wünschenswert, die übersetzten, imperativen Schleifenprogramme innerhalb von Haskell interpretieren zu können. Dazu wurde ein Interpreter entwickelt, der in Kapitel 8 beschrieben ist. Er wird unter Verwendung der Ersatzfunktion `accumarray` aufgerufen und liefert aus in Kapitel 8 dargelegten Gründen kein Array, sondern einen Aggregattyp als Rückgabewert. Diese Einschränkung ist aber nur für die Weiterentwicklung der Schleifensprache von Bedeutung.

Da bei der *lazy*-Semantik der Sprache Haskell nur wenige Aussagen darüber gemacht werden können, wann ein Ausdruck ausgewertet wird, muß die *comprehension* komplett gegen zu frühzeitige Auswertung geschützt werden. Dieses wird dadurch erreicht, daß die Parameterliste von `accumArray` als String an die Ersatzfunktion `accumarray` übergeben wird, der vom Compiler natürlicherweise nicht verändert wird.

Diese Designentscheidung hat zur Folge, daß der schon erwähnte Scanner/Parser benötigt wird, welcher diesen Parameterstring in seine Komponenten zerlegt, bevor mit der Auswertung begonnen werden kann. Dieser Parser kann dann auch gleich so konstruiert werden, daß nicht beliebige *comprehensions* akzeptiert werden, sondern die Restriktionen der LooPo Eingabesprache beachtet werden. Der implementierte Parser garantiert die Einhaltung der oben beschriebenen Einschränkungen, die an zu parallelisierende `accumArray`-Aufrufe gestellt werden.

Die erkannten Parameter werden in einen Schleifensatz umgeformt, der im aktuellen Kontext interpretiert werden kann. Innerhalb der Datenstruktur, die den Schleifensatz darstellt, werden die Variablennamen als String repräsentiert. Da für die Interpretation auch die Werte der Variablen benötigt werden, muß zusätzlich die Menge der benötigten Variablen in einer Liste an `accumarray` weitergegeben werden.

Das aus diesen Überlegungen resultierende Aufrufschema für die Interpretation des Ausdrucks hat dann folgende Form:

(`accumarray` String Store)	
↓	`alparser`, `scanner`
Parameterliste (Typ: `AL`)	
↓	`aaTransform`
Schleifensatz (Typ: `Cmd`)	
↓	`interpret`
Aggregat (Typ: `Agg`)	

Kapitel 7

Der Übersetzer in die Schleifensprache

Aufgabe des Übersetzers ist es, Ausdrücke, die in ihrer Form `accumArray` Funktionsaufrufen in Haskell entsprechen, in die in Kapitel 5 besprochene Schleifensprache umzuformen. Der Typ des `accumArray` in Haskell 1.2 [HPJW92] ist:

```
accumArray ::
  (Ix a) => (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b
```

Die Quelle der Parallelität ist die Auswertung der als *array comprehension* geschriebene *association list* als paralleler Schleifensatz. Ein Ersatz der `accumArray` Funktion muß daher den Haskell-Compiler an der Expansion der *comprehension* hindern. Als Lösung wird der gesamte, oben beschriebene Ausdruck als Zeichenkette übergeben, die dann mit Hilfe eines Scanners/Parsers in ihre Bestandteile zerlegt und in ein Schleifenprogramm umgewandelt werden kann. Da es in rein-funktionalen Sprachen auf oberster Ebene der Funktionsdefinitionen keinen globalen Zustand gibt, der bei Funktionsaufrufen automatisch mitgereicht wird, und daher alle von einer Funktion benötigten Parameter explizit übergeben werden müssen, benötigt die neue Funktion neben der Zeichenkette einen "Zustand", der die Werte der in der *comprehension* benutzten Variablen enthält. Der Typ von `accumarray` ist daher:

```
accumarray:: String -> Store -> Agg
```

7.1 Scanner und Parser

Innerhalb der `accumarray`-Funktion wird die oben übergebene Zeichenkette mit Hilfe eines Scanners und Parsers in den algebraischen Datentyp `AA` umgewandelt, aus dem dann später die Funktion `aaTransform` ein Schleifenprogramm erzeugt. Der Typ `AA` hat die Definition:

```
data LoopHead = LH Var Exp Exp
data AssocList = AL Index Exp [LoopHead]
data AA = AA (Exp -> Exp -> Exp) Val (Index,Index) AssocList
```

Die Reihenfolge und Bedeutung der Parameter in `AA` entspricht genau der in `accumArray`. Zuerst kommt die akkumulierende Funktion, danach das neutrale Element, Ober- und Unterindexgrenzen sowie die *association list*, die ein einem eigenen Typ ausgelagert ist. Der Typ `AssocList` besteht aus einem Index/Ausdruck Paar, von dem für jeden generierten Index eine Instanz in die Liste eingefügt wird. Schleifenköpfe, die im Typ `LoopHead` repräsentiert werden, entsprechen genau den Indexgeneratoren der *qualifier list* einer *array comprehension*. Sie enthalten die Schleifenvariable sowie Ober- und Untergrenze der Schleife. Im folgenden sei ein Beispiel für die gewünschte Transformation durch Scanner und Parser von einer `accumarray`-Parameterliste in einen Ausdruck vom Typ `AA` gegeben:

```
((alparser . scanner)
   "(+) 0 ((0),(16)) [ (i*j) := i+j | i <- [0..4], j <- [0..4]]")
```

Dieser Ausdruck kann z.B. in dem Interpreter Hugs ausgewertet werden und liefert als Ergebnis:

```
AA (:+:) (I 0) ((Ix1 (IConst 0)),(Ix1 (IConst 16)))
  (AL (Ix1 (Content "i" :*: Content "j")) (Content "i" :+: Content "j")
    [ LH "i" (IConst 0) (IConst 4), LH "j" (IConst 0) (IConst 4)])
```

Scanner und Parser wurden mit Hilfe des Parsergenerators *Happy* [GM96] erzeugt. Das Eingabeformat von Happy hat große Ähnlichkeit mit dem unter UNIX verbreiteten yacc(1). Es werden eine EBNF-Grammatik der Eingabesprache sowie die in den Semantikaktionen verwendeten Datentypen benötigt, aus denen Happy einen LALR(1) Parser als Haskell-Modul erzeugt.

An dieser Stelle werden nur die Produktionsregeln für `AA`-Ausdrücke erläutert. Die vollständige Datei inklusive Scanner-Funktion befindet sich in Anhang A. Zuerst werden die drei "einfachen" Parameter der `accumarray`-Funktion, der Operator, das Nullelement sowie das Indexpaar, erkannt:

```
 AccumArray :: { AA }
 AccumArray :
     '(' Op ')' Val '(' Idx ',' Idx ')' AList {AA $2 $4 ($6,$8) $10}
```

```
Op :: { Exp -> Exp -> Exp }
Op : '+'        { :+: }
   | '-'        { :@: }
   | '*'        { :*: }

Val :: { Val }
Val  : int                     { I $1 }
     | float                   { F $1 }

Idx :: { Index }
Idx : '(' Expr ')'             { (Ix1 $2) }
    | '(' Expr ',' Expr ')'    { (Ix2 $2 $4) }
```

Dann folgt die etwas aufwendigere *association list*:

```
AList :: { AssocList }
AList : '[' Idx ":=" Expr '|' GenList ']'  { AL $2 $4 $6 }

GenList :: { [LoopHead] }
GenList : Gen ',' GenList       { $1 : $3 }
        | Gen                   { [ $1 ] }

Gen : var "<-" '[' Expr ".." Expr ']'  {  LH $1 $4 $6 }
```

Jedem Generator entspricht später der Kopf einer `FOR`-Schleife, daher wird er auch direkt in den Typ `LoopHead` überführt. Darauf folgt die Erkennung der arithmetischen Ausdrücke; hierbei ist `Expr` der Name der entsprechenden Produktion, `Exp` der Haskell-Typ des resultierenden Ausdrucks:

```
Expr :: { Exp }
Expr : Expr '+' Term         { $1 :+: $3 }
     | Expr '-' Term         { $1 :@: $3 }
     | Expr '>' Term         { $1 :>: $3 }
     | Expr "<=" Term        { $1 :<=: $3 }
     | Term                  { ($1) }

Term : Term '*' Factor       { ($1 :*: $3) }
     | Factor                { $1 }

Factor : Number              { $1 }
```

```
        | '(' Expr ')'         { ( $2 ) }
        | var '!' Idx          { (ArrContents $1 $3) }
        | var                  { Contents $1 }

 Number :: { Exp }
 Number  : int                  { IConst $1 }
         | float                { FConst $1 }
```

Scanner und Parser sind in zwei verschiedenen Funktionen untergebracht, die durch Funktionskomposition zusammengesetzt werden. Das Ergebnis ist die Funktion

```
(alparser . scanner) :: String -> AA
```

Angewandt auf das obige Beispiel liefert der Aufruf

```
(aaTransform ((alparser . scanner)
   "(+) 0 ((0),(16)) [ (i*j) := i+j | i <- [0..4], j <- [0..4]]"))
```

das Ergebnis

```
For "i" (IConst 0) (IConst 4)
  (For "j" (IConst 0) (IConst 4)
     (ArrAssign "_result" (Ix1 (Contents "i" :*: Contents "j"))
          (ArrContents "_result" (Ix1 (Contents "i" :*: Contents "j"))
          :+: (Contents "i" :+: Contents "j"))
     )
  )
```

7.2 Die Übersetzung ins LooPo-Eingabeformat

Die Schleifensprache wird standardmäßig in C übersetzt, obwohl in dieser Arbeit für LooPo auch eine FORTRAN Schnittstelle implementiert wurde. Der Grund für die Wahl von C als Standard besteht in der Möglichkeit des Glasgow Haskell-Compilers, bei der Übersetzung von Haskell-Programmen Zwischencode in C zu erzeugen, zu dem dann eigene, direkt in C geschriebene Funktionen hinzugefügt werden können. Auf diese Weise können Berechnungen aus Haskell nach C verlagert werden. Die Funktion `c` wandelt einen Ausdruck vom Typ `Cmd` in ein fertig formatiertes C-Programm um, welches als Zeichenkette zurückgegeben wird. Die Transformation der des vom Parser zurückgegeben `AA`-Ausdrucks in den Typ `Cmd` liefert die in Abschnitt 7.1 erwähnte Funktion `aaTransform`. Zusammengesetzt erhält man analog zu oben aus

```
c (aaTransform ((alparser . scanner)
    "(+) 0 ((0),(16)) [ (i*j) := i+j | i <- [0..4], j <- [0..4]]"))
```

den C-Schleifensatz

```
for(i = 0; i<=4; i++) {
  for(j = 0; j<=4; j++) {
    _result[i * j] = _result[i * j] + i + j;
  }
}
```

Die Transformation in einen C-Schleifensatz wird bei der Generierung der Haskell-C-Schnittstelle (Kapitel 10) durchgeführt. Dort wird der Schleifensatz in eine C-Funktion eingebettet, welche zusätzlich die Parameter- und Ergebnisübergabe zum Haskell-Hauptprogramm erlaubt.

Kapitel 8

Der Interpreter für die Schleifensprache

Während der Entwicklung der Schleifensprache war eine Umgebung notwendig, in der die Sprache direkt getestet werden konnte. Desweiteren mußte die Korrektheit des Übersetzers in die Schleifensprache in einer Laufzeitumgebung überprüft werden. Daraus ergab sich der Wunsch nach einem Interpreter für die Schleifensprache, da während der Programmentwicklung Fehler am besten in der interaktiven Umgebung eines Haskell-Interpreters erkannt werden. Außerdem ist ein solcher Interpreter bei späteren Erweiterungsarbeiten an der Schleifensprache von Nutzen.

Um den Interpreter zu verwenden, soll das Haskell-Programm möglichst wenig verändert werden müssen. Die Haskell-Funktion `accumArray` wird dabei durch die Interpreter-Funktion `accumarray` ersetzt, welche als ersten Parameter die gesamte `accumArray`-Parameterliste in Form einer Zeichenkette bekommt. Diese Zeichenkette wird von Scanner und Parser in die im letzten Kapitel besprochene interne Darstellung umgeformt, die dann in das imperative Schleifenprogramm übersetzt wird, um interpretiert zu werden. Dabei gehen aber die Namensbindungen der in den Parameterausdrücken verwendeten Variablen verloren, so daß als weiterer Parameter eine Liste der verwendeten Variablen explizit übergeben werden muß.

Der Rückgabetyp ist nicht `Array a b` wie bei `accumArray`, sondern ein Aggregattyp, in dem Arrays der einzelnen Dimensionalitäten explizit durch verschiedene Konstruktoren bezeichnet werden. Der Grund hierfür wird im nächsten Abschnitt erläutert. Somit hat die Interpreter-Funktion `accumarray` den Typ

```
accumarray:: String -> Store -> Agg
```

wodurch bei ihrer Verwendung eine Typenanpassung im Quellprogramm erforderlich wird.

Für diese Änderungen gibt es aber gute Gründe, und da der Benutzer, der die Schleifensprache nicht weiterentwickeln möchte, den Interpreter nicht verwendet, ist diese Einschränkung tragbar.

Der verwendete Interpreter teilt sich in eine Funktion zum Auswerten arithmetischer Ausdrücke (`evalexp`) und die eigentliche Interpretation der Schleifenanweisungen (in der Funktion `interpret`). Die Grundstruktur entspricht üblichen Interpretern in funktionalen Sprachen, wie man sie in Standardlehrbüchern wie z.B. [Rea89] findet.

8.1 Der globale Zustand des imperativen Programms

Im Gegensatz zu funktionalen Programmen besitzen imperative Programme einen globalen Zustand, der sich durch die Belegung sämtlicher Variablen auszeichnet. In Sprachen mit Prozedur-/Funktionsaufrufen kommen gegebenenfalls noch die Belegung von im aktuellen Gültigkeitsbereich “verdeckten” Variablen sowie der Inhalt des Stapels mit Rücksprungadressen von Funktionsaufrufen hinzu.

Ein solcher Zustand wird im vorliegenden Fall von dem Datentyp *store* modelliert, der aus einer Liste von Paaren besteht: jeweils einem Variablennamen und dem zugehörigen Aggregat, welches beliebigdimensionale Arrays des Datentypes `Val` beinhaltet.

```
data Agg = A0 Val
         | A1 (Array Int Val)
         | A2 (Array (Int,Int) Val) deriving (Text)

type Var = String

data Store = Sto [(Var, Agg)]
```

An dieser Stelle müssen einige Designentscheidungen angesprochen werden, die grundsätzlicher Art sind und mit ähnlich guten Begründungen auch anders hätten getroffen werden können.

Durch den Typ `Agg` werden die Arraydimensionalitäten explizit aufgezählt; hierbei handelt es sich um das gleiche Problem wie schon bei dem Datentyp `Exp`, der in Abschnitt 5.2.1 beschrieben wurde. Die Dimensionalität des Arrays läßt sich nicht parametrisieren, d.h. man kann in Haskell keine Funktion angeben, die in der Dimension eines Arrays (ein Integer-Tupel) als Parameter polymorph ist. Da aber Arrays verschiedener Dimensionalitäten von ein und derselben Funktion verarbeitet werden sollen (die Alternative ist die entsprechenden Funktionen für jede Dimension unter anderem Namen zu implementieren), werden sie in `Agg` “eingebettet”.

Diese Einschränkung ist aber nicht schwerwiegend, da sämtliche Stellen im Quellcode, die explizit Konstruktoren von `Agg` aufzählen, als solche gekennzeichnet sind und auf mechanische Weise beliebig um zusätzliche Dimensionen erweitert werden können. Eine Alternative wäre eine Repräsentation der Dimensionstupel als Listen, wodurch allerdings die Verwendung von Haskell-Arrays verhindert würde, da Arrays über (Integer-)Tupel indiziert werden. Hilfsfunktion, die Listen in n-Tupel umwandeln, müßte, dann doch die Dimensionen aufzählen, wodurch das obige Problem nur verlagert worden wäre. Insgesamt hätte sich dieser Weg auf Umfang und Lesbarkeit des Interpreters sehr negativ ausgewirkt und ist daher nicht gewählt worden.

Ein weiterer, möglicher Kritikpunkt ist die Ausgliederung des Typs `Val` aus `Agg`, d.h. die Aufzählung der Werttypen in einem eigenen Datentyp und nicht in `Agg`. Dadurch ist z.B. die Definition von Arrays mit gemischten Typen möglich, was sicherlich nicht wünschenswert ist. Der Grund, warum dieser Weg trotzdem getroffen wurde, ist, daß es ansonsten zu einer erheblich höheren Anzahl Konstruktoren in `Agg` gekommen wäre, nämlich dem Produkt der Anzahl der Konstruktoren von `Agg` und `Val`. Weiterhin müssen einige Funktionen eine Fallunterscheidung über alle Konstruktoren von `Agg` machen; dadurch wäre das Programm unnötig kompliziert worden. Alternativ hätte man `accumarray` einen weiteren Parameter mitgeben können, der den Konstruktor des ausgewählten `Val`-Typen für das ganze Array festlegt. Diese Änderung hatte die Funktion aber noch weiter von der Originalsignatur von `accumArray` entfernt. Da aber nur Entwickler diese Funktion überhaupt benutzen, ist der gefundene Kompromiß tragbar.

Der Zugriff auf den globalen Speicher wird über Zugriffsfunktionen vorgenommen, die in der Regel paarweise (ohne Index für Skalarvariablen und mit Index für Arrays) existieren. Mit den bisherigen Erklärungen der verwendeten Datentypen wird die Semantik der verwendeten Funktionen schon allein durch ihren Typ ersichtlich:

```
ar_fetch :: Var -> Index -> Store -> Val
fetch :: Var -> Store -> Val
ar_update :: Var -> Index -> Val ->  Store -> Store
update :: Var -> Agg -> Store -> Store
initarray :: Var -> Index -> [Val] -> Store -> Store
```

So sind an erster Stelle die Lesefunktionen zu nennen, die den Wert von Variablen aus dem Speicher zurückliefern. Dieses Paar von Funktionen lautet `ar_fetch` und `fetch` – bei gegebenem Speicher, Variablennamen und ggf. dem Arrayindex wird der zugehörige Wert zurückgeliefert.

Demgegenüber steht das Paar von Schreibfunktionen, die den Wert einer Variablen im Speicher ändern. Die Namen sind analog `ar_update` und `update`. Beide Funktionspaare stützen sich auf einige Hilfsfunktionen, die die Implementation des Speicherzugriffs etwas besser strukturieren. Da diese Funktionen aber nur von den oben genannten verwendet werden wird auf ihre Beschreibung an dieser Stelle verzichtet.

Um Arrays überhaupt verwenden zu können, werden sie von der Funktion `initarray` mit Hilfe eines Indexes und einer Liste von `Vals` definiert. Skalarvariablen werden beim ersten Beschreiben automatisch angelegt.

In sämtlichen Funktionen, die einen Index als Parameter haben, muß eine Fallunterscheidung bezüglich der Dimensionalität des Indexes durchgeführt werden.

8.2 Die Auswertung von arithmetischen Ausdrücken

Einer der Hauptgründe für die Verwendung von algebraischen Datentypen zur Modellierung der Schleifensprache und der arithmetischen Ausdrücke war die vergleichsweise einfache Implementation der Interpretation. Die Funktion

```
evalexp :: Exp -> Store -> Val
```

wertet einen Ausdruck im Kontext des übergebenen Speichers aus und liefert das Resultat vom Typ `Val` zurück, dabei wird eine Fallunterscheidung über sämtliche Konstruktoren des Datentyp `Exp` durchgeführt.

Beispielhaft soll die Berechnung der Summe von zwei Summanden beschrieben werden. Die zugehörige Teildefinition von `evalexp` lautet:

```
evalexp (exp1 :+: exp2) s
    | (isint e1) && (isint e2)       = I (iexp1 + iexp2)
    | (isdouble e1) && (isdouble e2) = D (dexp1 + dexp2)
    | otherwise                      = error ("Plus: op types not equal.")
        where e1 = (evalexp exp1 s)
              e2 = (evalexp exp2 s)
              (I iexp1) = e1
              (I iexp2) = e2
              (D dexp1) = e1
              (D dexp2) = e2
```

Die Definition wird durch den Test, ob beide Operanden vom gleichen Typ sind, wesentlich komplizierter. In einer Sprache mit nur einem Typ sähe die Definition wie folgt aus:

```
evalexp (exp1 :+: exp2) s = (evalexp exp1 s) + (evalexp exp2 s)
```

Die Funktionen `isint` und `isfloat` akzeptieren einen Parameter vom Typ `Val` und liefern einen Wahrheitswert zurück. Somit kann über die verwendeten *guards* der richtige Typ des Ergebnisses gewählt werden. Man beachte, daß Haskell wegen der *lazy*-Semantik nicht alle Ausdrücke des `where`-Konstruktes auswertet (was unweigerlich zu einem Typfehler führen würde), sondern nur die tatsächlich benötigten.

8.3 Die Interpretation von Anweisungen

Die in Abschnitt 5.2.2 vorgestellte Schleifensprache basiert auf dem Datentyp `Cmd`, aus dessen Ähnlichkeit zum Typ `Exp` eine vergleichbare Interpretation resultiert. Die Funktion

```
interpret :: Cmd -> Store -> Store
```

erwartet als Parameter eine Anweisung und den aktuellen Speicher und liefert einen durch Auswertung der Anweisung veränderten Speicher zurück.

Die Funktionsweise von `interpret` soll an dem Beispiel der `WHILE`-Schleife verdeutlicht werden.

```
interpret (While e c) s
    = interpret (if (evalexp e s) == (I 1) then (c :>> (While e c))
                                           else (Nop              )) s
```

Der Typkonstruktor `WHILE` im Typ `Cmd` hat zwei Argumente: einen Ausdruck und eine Anweisung. Der Ausdruck erfüllt die Funktion der Schleifenbedingung, die bei einer Auswertung zum Integer-Wert 1 die Abarbeitung des Schleifenrumpfes bewirkt. Die Darstellung von Wahrheitswerten durch Integers ist aus Gründen der Vereinfachung gewählt worden und kann toleriert werden, da boolesche Ausdrücke nicht intern generiert werden, so daß der Benutzer nicht nach Art von Sprachen wie C mit solchen booleschen Integers rechnen kann.

Der Schleifenrumpf wird in der Anweisung übergeben, die unter Verwendung des `:>>` Operators wiederum an mehreren Anweisungen zusammengesetzt sein kann. Der Rumpf selbst wird aber nicht nur einmal ausgeführt, sondern es schließt sich die erneute Abarbeitung der `WHILE`-Schleife nach bekanntem Rekursionsmuster an.

Falls die Schleifenbedingung einen Wert ungleich 1 hat, wird eine leere Anweisung im `else`-Zweig ausgeführt, deren Ergebnis die Identität auf dem Speicher ist.

Auch für die weiteren Anweisungen der Schleifensprache (`:>>`, `IF`, `FOR`, `Assign`, `ArrAssign`, `DefArray` und `InitArray`) existieren Implementationen von `interpret`; sie werden über den Typkonstruktor des Datentyps `Cmd` — nichts anderes ist das Schlüsselwort — unterschieden.

Zum Abschluß sei die Funktion `interpret` an einem kleinen Beispiel verdeutlicht. Dazu werden zwei Hilfsfunktionen benutzt. `store0` liefert einen leeren Speicher zurück und `lookup` sucht eine Variable in einem Speicher und gibt den Aggregatwert zurück.

```
store0 :: Store
lookup :: Var -> Store -> Agg
interpret :: Cmd -> Store -> Store
```

Die Funktion `interpret` wird mit einem leeren Speicher und einem Programm aufgerufen, welches in diesem Speicher zwei Variablen definiert. Sie gibt den neuen, die beiden Variablen enthaltenden Speicher an `lookup`, welche den Wert der Variablen `A` ausgibt.

```
 lookup "A"
   (interpret
      ((Assign "B" (IConst 7)) :>>
       (Assign "A" ((IConst 5) :*: (Contents "B"))))
      store0)
```

Das Ergebnis bei Eingabe obiger Anfrage im Interpreter Hugs ist:

```
A0 (I 35)
```

Es handelt sich also um ein nulldimensionales Array (mit anderen Worten um eine Skalarvariable), deren Inhalt ein Integerwert von 35 ist.

Kapitel 9

Monaden in der funktionalen Programmierung

Der verstärkte Gebrauch von rein-funktionalen Sprachen wie Haskell in den achtziger Jahren warf die Frage auf, wie man imperative Sprachelemente, die "nicht rein-funktionale" Sprachen wie ML in einigen Bereichen überlegen wirken lassen, semantisch korrekt einführen könnte. Die Möglichkeit der Programmierung mit Seiteneffekten, globalen Variablen und *exceptions* machten einige Probleme in ML erheblich einfacher handhabbar als in jeder reinen Sprache. Da man mathematische Eigenschaften wie *referential transparency* auch in rein-funktionalen Sprachen mit einer *lazy*-Semantik nicht aufgeben wollte, konnte man die "unreinen" Konstrukte nicht direkt einbauen.

Hilfe kam aus der Kategorien- und Typtheorie, als Moggi [Mog90] zeigte, wie das Konzept einer "starken Monade" benutzt werden kann, um die denotationelle Semantik durch Einführung des "Berechnungkonzepts" zu strukturieren. Im nächsten Schritt machte Wadler [Wad90, Wad92a] dieses Konzept in der funktionalen Programmierung populär, in dem er praktikable Möglichkeiten aufzeigte, wie man die oben angesprochenen Sprachelemente in rein-funktionalen Sprachen mit Hilfe von Monaden modellieren kann.

Ein wichtiger Bereich für den Einsatz von Monaden ist die rein-funktionale Ein-/Ausgabe. Dies zeigt sich in der Tatsache, daß in dem aktualisierten Sprachstandard Haskell 1.3 [PHe96] die bisherigen Ein-/Ausgabemodelle (*stream based* und *CPS*) durch ein monadisches Konzept ersetzt wurden.

Im allgemeinen dienen Monaden an dieser Stelle drei Zwecken:

- um funktionale Programme zu strukturieren,
- um einen imperativ-ähnlichen Zustand zu modellieren und trotzdem die Beweisbarkeit von Eigenschaften funktionaler Ausdrücke sicherzustellen,

- um Programmierung allgemein zu vereinfachen.

Im folgenden werden Ergebnisse aus jüngerer Zeit aus der Arbeit mit Monaden in der funktionalen Programmierung präsentiert, um eine Idee zu vermitteln, wie die hochgesetzten Ziele erreicht werden könnten. Die Forschung auf diesem Gebiet ist noch nicht abgeschlossen. Weitere Details können in den zitierten Arbeiten nachgelesen werden.

Zu Beginn steht in Abschnitt 9.1 ein motivierendes Beispiel in Form der Haskell IO Monade. Danach folgt eine informelle Beschreibung der Monadenstruktur in der Kategorientheorie und ihrer Verbindung zur funktionalen Programmierung. Abschnitt 9.3 präsentiert Moggis Arbeit über das "Berechnungskonzept" und ihre Implementation durch Monaden. In einer erneuten Behandlung der IO Monade werden ihre Eigenschaften am Beispiel der "Zustandsübergangsmonade" beschrieben. In diesem Zusammenhang werden einige allgemeine Bemerkungen über monadisches Programmieren und dem Unterschied zu SML gemacht. Abschnitt 9.4 beschreibt die syntaktischen Ähnlichkeiten zwischen monadischer und imperativer Programmierung. Im Ausblick wird auf neuere Entwicklungen der Forschung eingegangen.

9.1 Beispiel: die IO Monade in Haskell 1.3

Dieser Abschnitt beschreibt die bekannte IO Monade aus Haskell 1.3 als motivierendes Beispiel für die folgenden Abschnitte. Ein Großteil der Beispiele ist [Wad95] entnommen. In der allgemein verbreiteten Darstellung bestehen Monaden in der funktionalen Programmierung aus einem Typkonstruktor und zwei darauf operierenden Funktionen, die ein paar Gesetzmäßigkeiten erfüllen müssen. In unserem Beispiel handelt es sich um den Typ `IO a`, der eine Berechnung beschreibt, die nach ihrer Auswertung ein Ergebnis vom Typ `a` zurückliefert. Eine nützliche Intuition ist es, sich Ausdrücke vom Typ `IO a` als Berechnungen vorzustellen, die nach Durchführung von Ein-/Ausgabeoperationen einen Wert zurückliefern. Dieses geschieht erst zu dem Zeitpunkt, wenn die Berechnung tatsächlich durchgeführt wird, der nicht notwendigerweise eintreten muß. Nach den Vorstellungen von Moggi sollte man also im Zusammenhang mit Monaden eher an Berechnungen als an Werte denken. Dieser Unterschied wird im folgenden klarer werden.

Wie bekommt man jetzt so eine "Berechnung"? Eine der eingangs erwähnten Funktionen ist:

```
return :: a -> IO a
```

die als Argument einen Wert vom Typ `a` nimmt und eine Berechnung zurückliefert (in diesem Fall die Identität), die bei ihrer Ausführung ihrerseits einen Wert vom Typ `a` ergibt. Diese Funktion wird auch oft `unit` genannt.

Die andere benötigte Funktion ist unter verschiedenen Bezeichnungen bekannt, deren verbreiteste `bind` ist, es kommen aber auch `>>=`, `*` und `then` vor. Sie stammt vom Kleisli-Stern ab (siehe Abschnitte 9.2) und dient der Verbindung zweier monadischen Berechnungen. Man kann sie sich als einen Kompositionsoperator ähnlich dem imperativen Semikolon vorstellen, wobei zusätzlich das Ergebnis der linksseitigen Berechnung an die rechte Seite durchgereicht wird:

```
>>= :: IO a -> (a -> IO b) -> IO b
```

Das Tripel (`IO`, `return`, `>>=`) wird *IO Monade* genannt, falls es die drei in Abschnitt 9.2 vorgestellten Regeln erfüllt. Der Zweck dieser Monade sei an einem Beispiel veranschaulicht. Mit Hilfe der Funktion

```
getc :: IO String
```

die eine Berechnung zurückliefert, welche sich zu einer Zeichenkette ergibt, kann man folgende Funktion `gets` definieren:

```
gets :: Int -> IO String
gets 0      = return []
gets (i+1)  = getc >>= \c -> gets i >>= \cs -> return (c:cs)
```

Offensichtlich handelt es sich bei `getc` um eine UNIX-ähnliche Funktion, die ein Zeichen einliest, während `gets i` die Eingabe einer Zeichenkette der Länge `i` erwartet (eine Zeichenkette in Haskell ist eine Liste von Zeichen).

Zuerst wird ein Zeichen eingelesen und in der dem `bind` folgenden λ-Abstraktion an den Bezeichner `c` gebunden. Danach wird `gets` rekursiv aufgerufen, um eine um ein Zeichen kürzere Zeichenkette zu lesen, wobei diese dann an `cs` gebunden wird. Zum Schluß wird das Zeichen `c` an die Liste `cs` angehängt und als Monade zurückgegeben.

9.2 Von der Kategorientheorie zur funktionalen Programmierung

Im vorangegangenen Abschnitt wurde eine Monade als Tripel (`IO`, `return`, `>>=`) eingeführt, wobei `IO` ein Typkonstruktor und `return` und `>>=` Funktionen mit einem bestimmten Typ sind. Wie schon erwähnt wurde, stammen Monaden aus der Kategorientheorie und werden dort auch schon seit einigen Jahrzehnten benutzt. Zu Beginn wurden sie *Standardkonstruktion* genannt, später folgte der mehrdeutige Begriff *Tripel*, der auch heute noch in einigen

Lehrbüchern zu finden ist. Mittlerweile hat sich aber der Begriff *Monade* (engl. *monad*) durchgesetzt, welcher an einen vom Leibniz eingeführten Begriff in der Philosophie angelehnt ist. In der funktionalen Programmierung wird eine spezielle Variante der allgemeinen Monade benutzt, nämlich die *starke Monade über einer kartesisch-abgeschlossenen Kategorie* [Mog90].

Dieser Abschnitt dient zwei Zielen: zum einen soll eine Brücke zwischen funktionaler Programmierung und Kategorientheorie geschlagen werden, um denjenigen die Verbindung zwischen beiden zu verdeutlichen, die schon Kenntnisse in beiden Bereichen besitzen. Zum anderen soll Lesern ohne Vorkenntnisse in Kategorientheorie wenigstens verdeutlicht werden, daß es dort zwei verschiedene, aber gleichwertige Repräsentationen einer Monade gibt, die auch beide in der Praxis vorkommen.

9.2.1 Ein paar Bemerkungen zur Kategorientheorie

Wir wollen nur einige grundlegende Begriffe der Kategorientheorie einführen. Eine ausführlichere Darstellung mit Bezug auf funktionale Programmierung ist [HC94], für noch mehr Erklärung bieten sich Lehrbücher wie [Pie91, Mac71] an.

Eine Kategorie besteht aus zwei Teilen: *Objekten* und *Pfeilen* (engl. *arrows*, auch *Morphismen* genannt). In einer Kategorie $\mathcal{C}$ werden sie mit $Obj(\mathcal{C})$ and $Arr(\mathcal{C})$ bezeichnet. Für den Einstieg ist die Vorstellung von Objekten als Mengen und Pfeilen als Abbildungen von einer Menge in eine andere hilfreich. Daraus resultierende Probleme wie die Russellsche Antinomie werden in den Lehrbüchern ausführlich behandelt. Die Komposition von Pfeilen ist auf natürliche Weise definiert.

Ein wichtiger Bestandteil der Theorie ist der *Funktor*, der eine Abbildung von einer Kategorie in eine andere darstellt. Folglich besteht der Funktor $T : \mathcal{C} \to \mathcal{D}$ aus zwei Teilen: $T_{Obj} : Obj(\mathcal{C}) \to Obj(\mathcal{D})$ und $T_{Arr} : Arr(\mathcal{C}) \to Arr(\mathcal{D})$. Wenn man sich jetzt wieder $\mathcal{C}$ als Kategorie der Mengen und $\mathcal{D}$ als Kategorie den Potenzmengen vorstellt, dann bildet T_{Obj} jede Menge aus $\mathcal{C}$ auf eine Potenzmenge in $\mathcal{D}$ ab, während T_{Arr} die Pfeile aus $\mathcal{C}$ auf diejenigen in $\mathcal{D}$ abbildet. In der Sprache der funktionalen Programmierung ist T_{Arr} eine Funktion höherer Ordnung. Leider hat es sich in Lehrbüchern eingebürgert, das Symbol T für T_{Obj} und T_{Arr} zu benutzen.

Die nächste Definition auf dem Weg zur kategorientheoretischen Erklärung einer Monade wäre die *natürliche Transformation*, die an dieser Stelle aber zur Vereinfachung weggelassen werden kann.

Es folgt eine informelle Beschreibung einer Monade. Eine Monade über einer Kategorie $\mathcal{C}$ ist eine Tripel (T, η, μ), wobei T ein Funktor von $\mathcal{C}$ nach $\mathcal{C}$ ist und es sich bei $\eta : id_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ um zwei natürliche Transformationen handelt. Sie müssen das *monadische Assoziativgesetz* sowie die *Gesetze über die linke und rechte Identität* erfüllen. Auch wenn

Details an dieser Stelle nicht erschöpfend behandelt werden, so sollte doch klar werden, daß eine Monade aus vier Teilen besteht: den zwei Teilen des Funktors T, sowie den beiden Abbildungen μ und η.

Die `IO` Monade im letzten Abschnitt besteht nur aus drei Teilen: einem Typkonstruktor und zwei Funktionen. Diese Monade ist eine Repräsentation des *Kleisli Tripels*. Das Kleisli Tripel ist eine andere Konstruktion der Kategorientheorie, die zur Monade äquivalent ist. Beide sind mächtig genug, die jeweils andere auszudrücken. Unser Kleisli Tripel ist das Tripel $(T_{Obj}, \eta, \_^*)$, wobei der *Kleisli Stern* den Morphismus $f : A \to T(B)$ zu $f^* : T(A) \to T(B)$ "liftet".

Beide Definitionen, $(T_{Obj}, \eta, \_^*)$ und (T, η, μ), entsprechen der allgemeinen Monade. Die erste notwendige Erweiterung ist, sie über einer *kartesisch-abgeschlossenen Kategorie* zu betrachten. Auf dieses Detail soll hier nicht weiter eingegangen werden. Die zweite Erweiterung, die zur Betrachtung von Monaden in der funktionalen Programmierung notwendig ist, bezieht sich auf die *starke Monade*, welche die sogenannte *Stärkefunktion* hinzufügt. Dabei handelt es sich um die natürliche Transformation $t_{A,B} : A \times TB \to T(A \times B)$, die dazu benutzt wird, die funktionalen Äquivalente von μ und dem Stern zu definieren. Eine genauere Behandlung dieser Thematik findet man in [Mog90].

Diese Kleisli Definition paßt auf unsere `IO` Monade: T_{Obj} ist ein Typkonstruktor (hier: `IO`), `return/unit` ist eine Implementation von η und `>>=/bind` ist der Kleisli Stern zusammen mit $t_{A,B}$.

Die vorherige Definition der vierteiligen Monade hat auch eine Korrespondenz in der funktionalen Programmierung. T_{Obj} und η werden wie eben behandelt, μ korrespondiert zu eine Funktion mit der Bezeichnung `join` und T_{Arr} mit $t_{A,B}$ ergeben die Funktion `map`:

```
unit:: a -> M a
map :: (a -> b) -> (M a -> M b)
join:: M (M a) -> M a
bind:: M a -> (a -> M b) -> M b
```

Um es zu wiederholen: die wesentliche Aussage ist, daß es zwei äquivalente Definitionen einer Monade gibt – (`M`, `unit`, `bind`) and (`M`, `unit`, `map`, `join`). Um sich das noch einmal zu verdeutlichen, seien `map` und `join` mit Hilfe von `bind` definiert:

```
map f m  =  m ‘bind‘ (\a -> unit (f a))
join z   =  z ‘bind‘ (\m -> m)
```

Auch die andere Richtung ist möglich:

```
m ‘bind‘ m  =  join (map k m)
```

Eine andere Darstellung von `unit`, `map` und `join` – die *monad comprehensions* – werden an dieser Stelle nicht behandelt. Ihre Beschreibung findet man in [Wad90]. Im folgenden wird die Kleisli Repräsentation einer starken Monade über einer kartesisch-abgeschlossenen Kategorie und der Begriff “Monade” als Synonym dafür behandelt.

Die schon angesprochenen Regeln, die eine Monade erfüllen muß, sind:

Linke Einheit:	`unit a >>= (\b -> n)`	$\equiv$	`n[b:=a]`
Rechte Einheit:	`m >>= (\b -> unit b)`	$\equiv$	`m`
Assziativität:	`m >>= \a -> (n >>= \b -> o)`	$\equiv$	`(m >>= \a -> n) >>= \b -> o`

In der Regel der “Linken Einheit” kann die Variable `b` frei im Term `n` vorkommen. Damit die Assoziativität gilt, darf `a` frei in `n` (aber nicht frei in `o`) und `b` frei in `o` vorkommen. Die vierteilige Repräsentation einer Monade benötigt mehr Regeln, die in [Wad92b] nachgelesen werden können.

9.3 Monaden als Berechnungen

Im Abschnitt 9.1 wurde das “Berechnungskonzept” erwähnt, die durch eine Monade dargestellt wird. Dieser Begriff (*notion of computation*) geht auf Moggi [Mog90] zurück, der Monaden benutzt, um Denotationelle Semantik zu strukturieren. Seine Vorstellungen sollen jetzt kurz beschrieben werden.

Wenn man Programme in der Kategorientheorie modelliert, identifiziert die Denotationelle Semantik Programme mit Morphismen einer Kategorie, deren Objekte mit Typen gleichgesetzt werden. Moggi schreibt: “Mit einem *Berechnungskonzept* meinen wir die qualitative Beschreibung der Bedeutung eines Programms in einer bestimmten Programmiersprache, im Gegensatz zu der Interpretationsfunktion selbst”. Als Beispiel für ein “Berechnungskonzept” nennt er “Berechnungen mit Seiteneffekten, bei denen ein Programm eine Abbildung von einem Speicher in ein Paar, bestehend aus einem Wert und dem modifizierten Speicher, beschreibt”. Bei der Modellierung dieses Berechnungskonzepts ist es nützlich, sich ein Programm als eine “Funktion von Werten nach Berechnungen” vorzustellen.

Durch diesen kategorientheoretischen Blick auf Berechnungen konnte Moggi Eigenschaften von Monaden in der Denotationellen Semantik von Programmiersprachen benutzen. Wadler griff diese Ideen auf und beschrieb einen monadischen Ausdruck vom Typ `M a` als eine Berechnung, die, “wenn sie jemals ausgeführt wird, einen Wert vom Typ `a` zurückliefert”.

Natürlich ist man aber am Ende eher an den Ergebnissen einer Berechnung interessiert, als an der Berechnung selbst. Wenn nun Monaden Berechnungen modellieren, wie bekommt man die Ergebnisse?

Ein kleines Beispiel soll die Dinge etwas verdeutlichen. Der Operator `>>` entspricht `>>=` mit dem Unterschied, daß das Ergebnis der ersten Berechnung nicht an die zweite weitergeleitet wird. Aus diesem Grund ist der Typ `>>::IO () -> IO () -> IO ()`. Dieser Operator wird benutzt, wenn auf der linken Seite eine Ausgabeoperation steht, deren Rückgabewert nicht weiter interessant ist.

```
hello:: IO ()
hello = putStr "Hello " >> putStr "World."
```

Diese Funktion vom Typ `IO ()` wird, falls sie jemals ausgeführt wird, eine Ausgabeoperation durchführen und den leeren Wert zurückliefern. Man beachte, daß gerade in einer Sprache mit einer *lazy*-Semantik nicht jede im Programm auftretende Berechnung notwendigerweise ausgeführt wird. Daraus ergeben sich zwei Fragen: wann wird diese Funktion denn ausgeführt und wie kann die richtige Reihenfolge der Ausführung von Ein-/Ausgabeoperationen in einer *lazy*-Sprache garantiert werden?

Die Antwort auf die erste Frage ist einfach: Jedes Haskell Programm ist eine Funktion namens `main` vom Typ `IO ()` – ihr Aufruf startet das Programm. In dem Programm

```
main:: IO ()
main = hello
```

wird die Kontrolle an die Funktion `hello` weitergegeben, wo die monadischen Berechnungen dann *ausgeführt* werden, so daß eine Ausgabe erzeugt und der leere Wert zurückgegeben wird.

Die zweite Frage ist damit aber noch nicht beantwortet: Wie kann garantiert werden, daß eine *lazy*-Sprache wie Haskell nicht zuerst die Berechnung `putStr "World."` ausführt, so daß die Ausgabe des Programms anstelle von “`Hello World.`” dann “`World.Hello`” ist? Eine solche Reihenfolge ist deshalb denkbar, da die Auswertungsreihenfolge von Funktionsparametern in einer funktionalen Sprache aufgrund der Church-Rosser-Eigenschaft beliebig ist und immer zum gleichen Ergebnis führt, so daß die Reihenfolge daher implementationsabhängig variieren darf. Diese Frage kann am Beispiel der Zustandsübergangsmonade beantwortet werden, die einen imperativen Zustand modelliert und als Prototyp der `IO` Monade gesehen werden kann.

9.3.1 Beispiel: die Zustandsübergangsmonade

Einen globalen Zustand kann man als Menge aller Variablen und ihrer Werte zu einem bestimmten Zeitpunkt während der Ausführung eines Programms betrachten, der durch Variablen (Re)zuweisung fortlaufend geändert wird. Da rein-funktionale Sprachen keine globalen Variablen besitzen und eine Zuweisung nur in der Form der Bindung von Namen an Funktionsargumente existiert, gibt es hier keinen globalen Zustand dieser Art. Ein Weg, diesen Zustand zu simulieren, ist, die Zustandsinformation in einem Datentyp zu verpacken und als Parameter allen aufgerufenen Funktionen mitzugeben. Das ist zwar auch die Grundidee der Zustandsübergangsmonade, aber indem der Zustand innerhalb der Monade "versteckt" wird, ergibt sich eine klarere Programmstruktur, da Änderungen im Zusammenhang mit dem Zustand nur die Definition der Monade und nicht die Typen aller beteiligten Funktion beeinflussen. Eine weitere wichtige Eigenschaft von Monaden dieses Typs ist, daß sie die Sequentialisierung von Operationen *garantieren* können. Dieses Verhalten ist für die `IO` Monade von großer Bedeutung.

Die folgende Zustandsübergangsmonade wurde (leicht modifiziert) aus Kapitel 2.5 von [Wad92b] entnommen:

```
type   S a  = State -> (a, State)

unitS x      = \s0 -> (x, s0)
m `bindS` k  = \s0 -> let (x,s1) = m s0
                          (y,s2) = k x s1
                      in  (y,s2)
```

Der Typkonstruktor einer Monade ist oft eine Funktion – man erinnere sich, daß er zur zur Objektabbildung eines Funktors korrespondiert. In der Haskell Syntax ist `S a` ein polymorpher Typ, der, wenn er z.B. mit `Int` instanziiert wird, eine Abbildung von `State` auf ein Paar darstellt, welches aus einem `Int`- und einem `State`-Wert besteht. Informell nimmt diese Funktion den globalen Zustand, führt eine Berechnung aus und liefert das Ergebnis der Berechnung und den geänderten globalen Zustand als Paar zurück.

In diesem Fall nimmt die Funktion `unit :: a -> S a` einen Wert vom Typ `a` und als Teil der Typdefinition von `S a` einen Zustand und gibt beide als Paar zurück. Mit anderen Worten: `unit a` ist die *identische Berechnung*. Man beachte, daß Funktionen, die einen monadischen Typkonstruktor in ihrer Signatur haben, in der Regel weitere, "versteckte" Argumente haben.

Die interessantere Funktion ist `bind :: S a -> (a -> S b) -> S b`. Ihre Parameter `m` und `k` haben die Typen `S a` bzw. `a -> S b`. Zuerst wird `m` ausgewertet, dann das Ergebnis an `k` übergeben und schließlich erfolgt die Rückgabe des Berechnungsergebnisses von `k`.

Aufgrund der Definition von `S` muß eine Umgebung übergeben werden, in der `m` und `k` ausgewertet werden sollen. Daher fordert `` m `bind` k `` als weiteres Argument den Zustand `s0` und berechnet `m s0`, dessen Resultat an `(x,s1)` gebunden wird. Man beachte, daß in den Beispielfunktionen die Variablen `x` und `y` die Typen `a` bzw. `b` haben.

Bis jetzt sind der Ergebniswert `x` und der modifizierte Zustand `s1` definiert, die beide an die Funktion `k` übergeben werden. Dabei wird `x` als direkter Parameter gewertet, was zu der Berechnung eines Ausdrucks vom Typ `S b` führt, welcher zur Auswertung wiederum einen Zustand braucht, nämlich `s1`. Als Ergebnis erhalten wir wieder ein Wert-Zustand-Paar, welches gleichzeitig das Endergebnis der `bind` Funktion ist.

Wie in Abschnitt 9.1 gesehen, kann `bind` auf natürliche Weise Ein-/Ausgabeoperationen auf eine imperative Art verbinden. Nun kann man sich der Frage zuwenden, wie die Reihenfolge der Berechnungen festgelegt wird.

Mit den bisherigen Erklärungen wird auch diese Antwort einfach: die Datenabhängigkeiten auf den Zuständen sind die Garanten der Reihenfolge. Da Haskell eine *lazy*-Sprache ist, wäre zunächst ein Compiler denkbar, der versuchen könnte, den Ausdruck `k` vor `m` auszuwerten und somit die vom Programmierer beabsichtigte Reihenfolge umzudrehen. Das ist aber nicht möglich, da die Auswertung von `k` in Zustand `s1` stattfindet. Bevor also `k` erfolgreich berechnet werden kann, wird somit `s1` und damit die Auswertung von `m`, dessen Ergebnis `s1` ist, notwendig.

Monaden können also benutzt werden, Daten wie einen globalen Zustand weiterzureichen. Die Datenabhängigkeiten, die möglicherweise zwischen den weitergereichten Daten bestehen, können ausgenutzt werden, um eine bestimmte Reihenfolge zu garantieren. Die Monade selbst garantiert keine vordefinierte Reihenfolge!

9.3.2 Monadische Programmierung

Der übliche Weg, eine Monade zu benutzen, ist wie folgt: man braucht die Monade selbst in Form eines Typkonstruktors sowie Implementationen der Funktionen `unit` und `bind`. Diese Implementationen müssen innerhalb der Semantik der verwendeten Programmiersprache die Monadengesetze erfüllen. Zusätzlich werden noch einige Hilfsfunktionen definiert, die direkt auf der Monade operieren. Im Abschnitt 9.1 sind die Funktionen `putc` und `getc` Beispiele dafür. Nur diese Funktionen manipulieren zusammen mit den Monadenoperatoren direkt die interne Repräsentation der Monade, wie z.B. den Zustand.

Auf der nächst höheren Ebene des funktionalen Programms machen Funktionen Gebrauch von der Monade, in dem sie z.B. `return`, `>>=`, `getc` usw. benutzen. In diesem Zusammenhang ist der interne Zustand der Monade nicht mehr sichtbar und genau aus diesem Grund werden Monaden auch als *Strukturierungsmethode* für funktionale Programme bezeichnet. Sie fügen hier eine zusätzliche Abstraktionsebene ein.

Da funktionale Programme auf der Ebene der Funktionsdefinitionen keine global zugreifbaren Datenstrukturen besitzen, müssen alle benötigten Daten als Parameter durch die Aufrufhierarchie der Funktionen hindurchgereicht werden, während die Ergebnisse auf gleichen Weg wieder “hochgereicht” werden. Änderungen in den Datenstrukturen haben kaskadische Änderungen in großen Teilen des Codes zur Folge. Um das zu vermeiden, kann man die gesamte Parameterliste einer Funktion in einen monadischen Typkonstruktor einbetten, der dann als einziger Parameter stehenbleibt. Wenn jetzt Änderungen notwendig werden, müssen zwar immer noch die Monade und ihre Hilfsfunktionen auf den neuen Stand gebracht werden, aber die Anpassung der Parameterlisten und zum Teil auch der Rümpfe aller übrigen Funktionen fällt deutlich kleiner aus. Ein kleines Programmfragment von Will Partain, das diesen Programmierstil illustriert, ist im Quellcode des Glasgow Haskell Compilers (Version 0.26 /0.29) in der Datei `simple-monad.lhs` enthalten. Der Compiler benutzt diese Technik an vielen Stellen selbst.

9.3.3 Vorteil der Existenz von *equational reasoning* in rein-funktionalen Sprachen

Ein oft genannter Vorteil von rein-funktionalen Sprachen ist die Möglichkeit, durch sogenanntes Gleichungsbeweisen (engl. *equational reasoning*) Aussagen über Programmeigenschaften machen zu können. Monaden bewahren diese Eigenschaft, tatsächlich können die drei Monadengesetze sogar in Beweisen verwendet werden. Die Gegenwart von Seiteneffekten zerstört aber diese nützliche Eigenschaft.

Dabei ist zu beachten, daß der Begriff “Seiteneffekt” nicht eindeutig definiert ist. Diser Begriff umschreibt die Auswirkungen von Funktionen, die zusätzlich zu ihrem eigentlichen Rückgabewert geschehen. Beispiele dafür sind Ein-/Ausgabe sowie die Veränderung des globalen Zustands eines Programms. Die vorherrschende Lesart ist, daß Ein-/Ausgabe zwar auch in rein-funktionalen Sprachen ein Seiteneffekt ist, durch seine Kapselung in Monaden aber die *equational reasoning* Eigenschaft nicht verletzt. Um sich den Unterschied zu verdeutlichen, sei ein Beispiel aus [Wad95] angebracht. Hier wird demosntriert, daß die Anwesenheit von *equational reasoning* in Haskell zur Folge hat, daß das Programm die naheliegende und wahrscheinlich vom Programmierer beabsichtigte Ausgabe macht, während die Ausgabe des ML-Programms einiger Erklärung bedarf.

Die `output` Funktion in ML macht als Seiteneffekt eine Ausgabe und liefert den leeren Wert zurück. Also gibt

```
output(std_out, "ha"); output(std_out, "ha")
```

die Zeichenkette `haha` aus, während

```
let val x = output(std_out, "ha") in x; x end
```

nur `ha` schreibt und zweimal den leeren Wert zurückgibt. Dieses Verhalten ist möglicherweise vom Programmierer nicht beabsichtigt gewesen. Auf jeden Fall verletzt es die *referential transparency*, die fordert, daß *Gleiches gegen Gleiches* ersetzt werden darf. Im Fall von SML gestaltet sich die Durchführung von Korrektheitsbeweisen schwierig, da algebraische Gesetze wie z.B. `2 * a = a + a` nicht für alle Typen `a` gelten. Wenn man aber die beiden vorangegangenen Codestücke in Haskell übersetzt, dann haben sie immer die gleiche Wirkung:

```
puts "ha" >> puts "ha"
```

gibt tatsächlich das gleiche aus wie

```
let m = puts "ha" in m >> m
```

Der Grund dafür ist, daß der Ausdruck `puts "ha"` eine monadische *Berechnung* definiert, die erst ausgewertet werden muß, bevor eine Ausgabe geschieht. Durch die *lazy*-Semantik von Haskell erfolgt die Auswertung erst, wenn die Ergebnisse eines Ausdrucks auch wirklich gebraucht werden, d.h. wenn das Ergebnis von `m >> m` benötigt wird. In diesem Ausdruck stehen dann zwei Berechnungen, die in der richtigen Reihenfolge nacheinander ausgeführt werden, was in diesem Fall zwei Ausgabeoperationen zur Folge hat.

9.4 Monaden und imperative Programmierung

Inzwischen sollte klar geworden sein, daß der Gebrauch der `bind` Operation einem imperativen Semikolon ähnelt. Diese Ähnlichkeit ist beabsichtigt und erleichtert die Programmierung mit Monaden. Tatsächlich kann man so funktionale Programme schreiben, die auf syntaktischer Ebene imperativen sehr ähnlich sind. Dies zeigt ein Beispiel aus [Wad95] sehr deutlich, in der eine Funktion `echo`, die einfach nur die Standardeingabe auf die -ausgabe kopiert, in C und Haskell geschrieben wurde:

```
echo :: IO ()
echo = getc >>= \c ->
       if (c == eof) then
          done
       else
          putc c >>
          echo
```

Hier wurde eine kleine Erweiterung der in Abschnitt 9.1 vorgestellten `IO` Monade benutzt: `done:: IO ()` ist ein Spezialfall von `return`, welcher immer den leeren Wert zurückliefert.

Das äquivalente C Programm sieht wie folgt aus:

```
echo () {
loop:  getchar(c);
       if (c == eof) {
          return
       } else {
          putchar(c);
          goto loop;
       }
}
```

Ob man in der eigenen Programmierung diese Ähnlichkeiten noch forciert, hängt vom persönlichen Programmierstil ab.

9.5 Neuere Entwicklungen

Wir haben nur Monaden vorgestellt, die eine einzige Aufgabe wie z.B. Ein-/Ausgabe verrichten. Der nächste Schritt ist, mehrere Monaden mit verschiedenen Eigenschaften zu kombinieren. Zuerst wurden direkte Kombinationen in [KW92, JD93] behandelt, später führte man *monad transformers* ein, die auf einer höheren Abstraktionsebene eine Monade in eine andere mit weiteren Eigenschaften umformen ([LJH95]). Ein Vergleich von Monaden mit anderen Ansätzen, die alle unter dem Oberbegriff *mutable abstract data types* (*MADT*) zusammengefaßt werden, findet man in [Hud93]. Ein interessanter Überblick über die aktuellen Vorgänge in der funktionalen Programmierung ist [JM95].

Kapitel 10

Die Haskell/C Schnittstelle

Dieses Kapitel beschreibt die automatische Generierung der Schnittstellenfunktionen in C und Haskell, um die Berechnung des Schleifensatzes in C auszuführen und dazu die benötigten Daten aus dem Haskell-Programm an das C-Programm zu übergeben und das Ergebnis auf dem gleichen Weg wieder zurückzuliefern.

10.1 Grundlegendes

Während der in den vorangegangenen Kapiteln beschriebene Haskell-Code dem Sprachstandard 1.2 entspricht und somit von allen Compilern akzeptiert wird, handelt es sich bei der Schnittstelle zur Sprache C um nicht-standardisierte Erweiterungen des Glasgow Haskell-Compilers ghc. Die vorhandene Implementation wurde mit der Version 0.26 angefertigt und wird möglicherweise aufgrund von Änderungen der C-Schnittstelle auf späteren ghc-Versionen nicht ohne Anpassung laufen. Die Dokumentation der Erweiterungen befindet sich in der ghc-Distribution im *User's Guide* und in *GHC prelude: types and operations*.

Die Struktur der Schleifensprache, des Interpreters und des Übersetzers fordert keine Beschränkungen auf dem Werttyp des Ergebnisarrays, außer daß es sich um numerische Typen handeln muß. Aus Gründen der Vereinfachung wurde die Schleifensprache nur für die Typen `Int` und `Double` implementiert; der zur Erweiterung nötige Aufwand wurde in vorangegangenen Kapiteln beschrieben. Die Implementation der Haskell/C-Schnittstelle beschränkt sich auf die Verwendung von `Int`, dem in C der Typ `long int` entspricht. Auch hier ist eine Erweiterung von mechanischer Natur.

Für die zu implementierende Schnittstelle ist dabei die Funktion `__ccall__` von zentraler Bedeutung, die als Haskell-Funktion als ersten Parameter den Namen einer C-Funktion bekommt, welche mit den dann weiter folgenden Parametern aufgerufen wird:

$$\texttt{\_\_ccall\_\_}\ \textit{cfunction\_name}\ a_1 \ldots a_n$$

Dieser Funktion kann kein eindeutiger Typ im Sinne von Haskell zugeordnet werden, da die Anzahl der Parameter variabel ist. Dies kann nur durch eine direkte Implementierung im Compiler erreicht werden. Die Typen der Funktionsparameter müssen Instanzen der Klasse `ccallable` sein. Dadurch wird sichergestellt, daß nur solche Typen vom Haskell-Programm übergeben werden, die einfachen C-Typen wie z.B. `int`, `double`, `char` entsprechen. Es können zwar auch Zeiger übergeben werden, aber keine komplexeren Strukturen wie Listen, Arrays oder algebraische Datentypen. Eine vergleichbare Einschränkung gilt auch für den Rückgabetyp, der eine Instanz der Klasse `creturnable` zu sein hat.

Da es für die Schnittstelle zu der nach C transformierten `accumArray`-Funktion notwendig ist, auch Haskell-Arrays zu übergeben, muß ein Umweg eingeschlagen werden. Der Glasgow Haskell Compiler stellt zu diesem Zweck den Typ `ByteArray` zur Verfügung, dessen Speicher in Haskell schon alloziert wird und von dem dann nur ein Zeiger mittels `ccall` an C übergeben wird. Auf der C-Seite muß dann aus diesem `ByteArray` die eigentliche Datenstruktur wieder rekonstruiert werden. Ein wesentlicher Bestandteil beider Schnittstellenfunktionen ist daher die Konversion von Arrays in ByteArrays und wieder zurück.

Die Frage des Anschlusses einer imperativen Sprache mit Seiteneffekten an eine rein funktionale Sprache ist insbesondere nach den Erläuterungen in Kapitel 9 keine einfache Angelegenheit, wenn man Eigenschaften wie *referential transparency* erhalten möchte. An gleicher Stelle wurde auch schon erläutert, daß die Programmierung mit Monaden eine Lösungsmöglichkeit darstellt. Tatsächlich wurde das C-Schnittstelle im ghc-Compiler unter Verwendung der sogenannten `PrimIO` Monade implementiert, die der schon vorgestellten `IO` Monade ähnlich ist. Der oben vorgestellte `__ccall__`-Aufruf hat den Ergebnistyp `PrimIO` *result*, wobei *result* der Haskell-Typ des Rückgabewertes der C-Funktion ist. Das Äquivalent des `>>=` Operators ist bei `PrimIO` die Funktion `thenPrimIO`, dem `>>` entspricht `seqPrimIO` und das `return` heißt hier `returnPrimIO`.

10.2 Die C-Schnittstelle in Haskell

Für jeden auftretenden `accumArray`-Aufruf wird eine Haskell-Funktion erzeugt, die alle außerhalb des `accumArray`-Aufrufes gebundenen und in den Parametern benutzten Variablen selbst als Parameter erhält und das Ergebnisarray zurückliefert. Diese Schnittstellenfunktion bekommt den Namen `computeInC_`n, wobei n eine fortlaufende Numerierung darstellt. Innerhalb dieser Funktion werden die Haskell-Arrays in `ByteArray`s umgeformt, die C-Funktion mittels `__ccall__` aufgerufen und das erhaltene Ergebnis-`ByteArray` in ein Haskell-Array umgeformt.

Das allgemeine Schema ist in Abbildung 10.1 dargestellt.

```
computeInC_n::Array (..)_1 ->... ->Array (..)_l ->Int_1 ... Int_m ->Array (..)
computeInC_n array_1 ... array_l var_1 ... var_m =
  unsafePerformPrimIO (
    ⋮
     (newIntArray (lower_bd_i, upper_bd_i)) `thenPrimIO` \barray_i ->  }
     (packdimI barray_i array_i) `seqPrimIO`                           } 1 ≤ i ≤ l
    ⋮
    (newIntArray (lower_bd_res, upper_bd_res)) `thenPrimIO` \result ->

    (__ccall__ cfunction
              ... (barray_i upper_bd_i,1 ...) ...                 }1 ≤ i ≤ l
              (result upper_bd_i,1 ...) var_1 ... var_m) `seqPrimIO`

    (returnPrimIO
      (listArray (result_bounds)(unpackdimI result ( result_bounds ))))
  )
  where bound definitions
```

mit $1 \leq i \leq l$

Abbildung 10.1: Das Interface-Schema in Haskell

Innerhalb der `unsafePerformPrimIO`-Funktion steht eine Sequenz von monadischen Operatoren, eingeleitet von Befehlspaaren, die ein `Int-ByteArray` anlegen und das entsprechende Haskell hineinkopieren. Das neue, von `newIntArray` angelegte Array wird jeweils nach dem `` `thenPrimIO` ``-Ausdruck an einen Namen gebunden, den man sich als Zeiger auf dieses Array vorstellen kann. Der Aufruf der Funktion `pack`*dim*`I` mit *dim* als Dimension des Arrays kopiert dann zwar den Inhalt des Haskell-Arrays als Seiteneffekt in das neue `ByteArray`, der Zeiger bleibt aber unverändert und somit wird die *referential transparency* gewahrt. Insbesondere muß von dieser Operation auch kein Ergebnis weitergegeben werden, da der Name des neuen Arrays weiterhin gültig ist. Deshalb wird der Aufruf von `pack`*dim*`I` mit dem Operator `` `seqPrimIO` `` abgeschlossen, dem bekanntlich keine λ-Abstraktion folgt.

Für den Rückgabewert der C-Funktion ist der einfachste Weg, ein `ByteArray` entsprechender Größe schon vor dem Aufruf von `__ccall__` anzulegen und der Funktion als Parameter mitzugeben. Die C-Funktion wird dann so wie die bisherigen `pack`-Funktionen das Array als Seiteneffekt beschreiben, so daß kein Rückgabewert von `__ccall__` benötigt wird.

Nach dem `__ccall__`-Aufruf erfolgt dann nur noch das Kopieren des Ergebnisses in ein

Haskell-Array, welches dann als Funktionswert zurückgeliefert wird.

Die gesamte monadische Berechnung hat den Ergebnistyp `PrimIO (Array` *Indextyp* `Int)`. Um diese monadische Berechnung tatsächlich auszuführen, müßte sie, wie im vorherigen Kapitel beschrieben, mit der `IO`-Monade der `main`-Funktion verbunden werden. Nun werden aber die übergeordneten Funktionen des `accumArray`-Aufrufes im allgemeinen nicht von einem Monadentyp sein und eine solche Umformung hätte Typänderungen von weiten Teilen des Programms zur Folge, die automatisch nicht machbar wären. Da aber innerhalb der C-Funktion wirklich nur die Arrayberechnungen ohne Seiteneffekte stattfinden, kann man die ghc-Funktion `unsafePerformPrimIO` verwenden, die die Berechnung des monadischen Ausdrucks direkt auslöst und das nicht-monadische Ergebnis zurückliefert. Die Autoren des ghc warnen zwar davor, die Funktion zu verwenden, wenn man z.B. in der C-Funktion Seiteneffekte produziert, was aber bei der hier vorgestellten Schnittstelle nicht der Fall ist. Ein Beispiel für eine generierte Schnittstelle findet man in Kapitel 11.

10.3 Die Haskell-Schnittstelle in C

Im Vergleich zur Haskell-Seite der Schnittstelle ist die aufgerufene C-Funktion einfach. Die übergebenen Parameter wurden oben schon beschrieben. Zu Beginn der Funktion erfolgt ein Umkopieren der `ByteArrays` in entsprechend dimensionierte C-Arrays mit Elementen vom Typ `long int`. Dabei wird das Ergebnisarray mit dem `_null`-Wert vorbelegt, welcher genau das neutrale Element ist, welches der `accumArray`-Funktion als zweiter Parameter übergeben wurde. Darauf folgend wird der nach C transformierte, sequentielle Schleifensatz eingesetzt, in dem das Ergebnisarray berechnet wird. Dieses Ergebnis wird in das schon übergebene `ByteArray` für das Funktionsergebnis kopiert und zum Schluß wird noch ein später nicht benutzter dummy-Wert zurückgegeben, da der Typchecker des Haskell-Compilers `void`-Funktionen ablehnt.

Das Schema der C-Funktion befindet sich in Abbildung 10.2.

Auch zu dem C-Code existiert eine Reihe von Hilfsfunktionen zum dynamischen Anlegen von Arrays, Kopieren von Arrays, usw., die an den automatisch generierten Code angehängt werden. Sie sind alle einfach, weshalb auf ihre Beschreibung an dieser Stelle verzichtet wird.

10.4 Die automatische Codegenerierung

Nachdem alle Funktionen zur Transformation eines `accumArray`-Aufrufes vorhanden sind, muß das Mosaik nur noch zusammengesetzt werden, wobei der gesamte Ablauf möglichst automatisch vonstatten gehen soll. Zu einem gegebenen Haskell-Programm wird ein verän-

```
c_function_nr(
        ⋮
        StgByteArray barray_i, long upper_bd_i,1, ...            (1 ≤ i ≤ l)
        ⋮
        StgByteArray _bresult, long res_upper_bd_1, ...
        long var_1, ... ,long var_m, long _null){
  /* allocate mem for C arrays and copy input into them */
  ⋮
  long pointer_to_array = init_dimd_array( corresp_ByteArray, upper_bd_i,1, ...);
  ⋮
  long pointer_to_result = alloc_dimd_array( res_upper_bd_1, ... ,_null);
  long loop_variables ...

  for( ... ){
    the C loop nest
  };

  longdimdToByteArray( _bresult, result,upper_bd_1, ... );
  /*return bogus val*/
  return 0;
  }
```

Abbildung 10.2: Das Interface-Schema in C

dertes Haskell-Programm mit eingefügtem Schnittstellen-Code sowie ein C-Programm mit dem transformierten Schleifensatz und entsprechendem Schnittstellen-Code erzeugt. Zusammen übersetzt ergibt sich das gleiche Laufzeitverhalten wie beim Original-Programm.

Das in den Abschnitten 10.3 und 10.2 beschriebenen Verfahren zur Erzeugung des Schnittstellencodes wird vom Programm `intfc_codegen` implementiert, dessen Quellcode im Anhang C aufgeführt ist. Dieses Haskell-Programm erwartet als Eingabe die gesamte Parameterliste eines `accumArray`-Aufrufes und erzeugt die Schnittstellen-Funktionen für C und Haskell.

Um den gesamten Ablauf zu automatisieren, müssen folgende Tätigkeiten ausgeführt werden:

- Das Originalprogramm muß auf Vorkommen von `accumArray`-Aufrufen untersucht werden, deren Parameterlisten dann extrahiert werden.

- Diese Parameterlisten werden einzeln an `intfc_codegen` übergeben, um für jedes `accumArray` die Schnittstellen in fortlaufender Numerierung zu erzeugen.
- Das originale Haskell-Programm wird in eine neue Datei namens *name*`_intfc.hs` kopiert, wobei jeder `accumArray`-Aufruf durch den Funktionsaufruf der entsprechenden Schnittstellenfunktion ersetzt wird. Die Schnittstellenfunktion selbst wird an das Programm angefügt.
- Es wird eine C-Datei namens *name*`_cintfc.c` angelegt, die die C-Schnittstellen enthält.
- An beiden neuen Quelldateien werden die jeweils benötigten Hilfsfunktionen angehängt.
- Es wird eine C-Headerdatei namens *name*`_intfc.h` erzeugt, die die Funktionsköpfe der C-Schnittstellenfunktionen umfaßt. Diese Datei wird dem ghc in der Kommandozeile beim Übersetzen von *name*`_intfc.hs` mitgegeben, um die `ccall`-Aufrufe auf Typkonflikte überprüfen zu können

Diese Liste von Aufgaben wird von einem UNIX-Shellscript namens `mkCinterface` abgearbeitet, dem Dateiname des Original-Quellprogrammes in der Kommandozeile übergeben wird. Die Manipulation des Quellcodes stützt sich dabei auf die Editierkommandos `awk` und `sed`. Zur Erzeugung der eigentlichen Schnittstellen wird wiederholt `intfc_codegen` aufgerufen. Dieses Script befindet sich im Anhang D.

10.5 Ansteuerung eines Parallelrechners

Die aktuelle Version von LooPo gibt, wie auch in Kapitel 11 beschrieben, den reinen parallelisierten Schleifensatz aus. Im Rahmen einer anderen Diplomarbeit soll der gesamte Schnittstellencode für den Ablauf auf einem Parallelrechner erzeugt werden [Fab97]. Analog zur vorher beschriebenen Haskell/C-Schnittstelle sind zwei Teile notwendig: das Parallelrechnerprogramm, welches die parallelisierten Schleifen enthält, sowie der Aufruf, der im C-Programm die sequentiellen Schleifen ersetzt und den Parallelrechner ansteuert. Die geplante Implementierung umfaßt mehrere Rechnerarchitekturen; als Zielsystem werden insbesondere eine Parsytec GCel mit Parix [Par93] sowie ein Solaris Rechnerpool unter MPI [DHHW93] angestrebt.

Im folgenden sei diese Schnittstelle kurz beschrieben. Weitere Details können in [Fab97] nachgelesen werden.

Das Haskell-Programm kann mehrere Schleifenausdrücke enthalten, weswegen im Zielprogramm mehrere parallelisierte Schleifen existieren können. Für jede solche Schleife wird folgender Parallelrechnercode erzeugt. Er besteht aus Definitions- und Kommunikationsanweisungen sowie der eigentlichen Schleife:

```
[<DeklarationArray>]
[<DeklarationKonstante>]

[<andereFunktionen>]

<FunktionsKopf>
 [<LadeAnweisungArray>]
 [<LadeAnweisungKonstante>]
 <parallele Schleife>
 [<SchreibeAnweisungArray>]
<FunktionsEnde>
```

An diese Programmstücke wird dann ein Initialisierungsteil angehängt, der eine Symboltabelle mit den verwendeten Array- und Konstantennamen anlegt und auf Aufforderung vom Hostrechner den entsprechenden Schleifensatz ausführt.

```
<FunktionsKopf>
 LooPo_initSymtabs(<#arrays>,<#const>);
 [<LOOPO_DeklarationArray>]
 [<LOOPO_DeklarationKonst>]

 <VerbindungMitHost>
 switch(<LadeBefehl>)
 {
  case <Abbrechen>: <BrecheAb>;
                    break;
  [case <i>:<FunktionsAufruf<i>>break;]
 }
<FunktionsEnde>
```

Das eigentliche C-Programm wird derart modifiziert, daß zu Beginn in einer Initialisierung die Verbindung zum Parallelrechner aufgebaut wird, die Schleifen durch Aufrufe an den Parallelrechner ersetzt werden und zum Schluß die Verbindung wieder abgebrochen wird.

```
<Initialize>
...
<Parallelrechneraufruf>
...
<End>
```

Diese Schnittstelle könnte nach ihrer vollständigen Implementierung zu einer Erweiterung dieser Diplomarbeit führen, indem ein auf einer Workstation laufendes Haskell-Programm durch Parallelrechnerverwendung beschleunigt wird. Dazu wird das Haskell-Programm mit der hier entwickelten Haskell/C-Schnittstelle versehen und im C-Teil die sequentielle Schleife durch den oben beschriebenen Parallelrechneraufruf ersetzt. Beim Start des Haskell-Programms auf einer Workstation wird gleichzeitig das Parallelrechnerprogramm auf einem Parallelrechner gestartet. Zur Laufzeit werden die zur Schleifenberechnung benötigten Arrays und Konstanten mittels Socket-Kommunikation übergeben und das Berechnungsergebnis empfangen.

Kapitel 11

Fallbeispiel einer Parallelisierung

Das folgende Beispiel soll zwei Zwecke erfüllen: zum einen sollen die in den vorangegangenen Kapiteln beschriebenen Methoden im Zusammenhang veranschaulicht werden, zum anderen wird durch die Verwendung eines bekannten Beispiels aus der Schleifenparallelisierung die Vergleichbarkeit zu anderen Arbeiten hergestellt.

Die Wahl fiel auf das Matrixprodukt, einem dreifach verschachtelten Schleifensatz. Zuerst sei die Verwendung der Standardfunktion `accumArray` beschrieben, um die Ausgangssituation vor der möglichen Parallelisierung zu beschreiben. Hier werden die beiden 4x4 Matrizen `b` und `c` multipliziert und als Funktionswert zurückgeliefert.

```
matmulttest:: Array (Int,Int) Int
matmulttest =
    accumArray (+) 0 ((0,0),(3,3))
        [(i,j) := b!(i,k) * c!(k,j) | i <-[0..3], j<-[0..3], k<-[0..3]]
    where b = listArray ((0,0),(3,3)) [ 2,  5,   7,  9,
                                        -5, 10,  7,  3,
                                        3,  8,  23, -4,
                                        65, 11, 15,  1]
          c = listArray ((0,0),(3,3)) [23,  8,  1,  3,
                                       17,  2,  9,  6,
                                       -9,  2, -5, 12,
                                        1, -3,  4,  7]
```

Dabei ist zu beachten, daß sich die Parallelisierung nur auf den in der *comprehension* enthaltenen Algorithmus bezieht und die möglicherweise auch parallelisierbare Berechnung der Arrays `b` und `c` unberücksichtigt läßt.

Der Benutzer schreibt also sein Haskell-Programm mit `accumArray`-Aufrufen wie bisher, achtet aber auf die in Kapitel 6 erwähnten zur Parallelisierung notwendigen Einschränkungen. Sobald das Programm fertig entwickelt und getestet worden ist, kann mit der Parallelisierung begonnen werden. Dazu wird dem Shellscript `mkCinterface` das Haskell-Programm als Parameter übergeben, was dann beispielsweise so aussehen könnte:

```
bash$ mkCinterface aatest.hs
Replacement of accumArray expressions by ccalls to C

substitution  # 1
Starting to replace:
 (+) 0 ((0,0),(3,3)) [(i,j) := b!(i,k) * c!(k,j) | i <-[0..3], j<-[0..3],
k<-[0..3]]
Done.
```

Die identifizierten `accumArray`-Aufrufe werden mit Hilfe von UNIX Hilfsprogrammen extrahiert und dem Codegenerator (siehe Kapitel 10) als Eingabe übergeben. Dieser generiert den Haskell-Code für die Schnittstelle zu C; dieser *Interface*-Code wird in den Quellcode eingesetzt und unter dem Namen <*source*>`_intfc.hs` abgespeichert.

In diesem Beispiel sieht das Ergebnis so aus:

```
matmulttest:: Array (Int,Int) Int
matmulttest = (computeInC_1 b c)
           where b = listArray ((0,0),(3,3)) [ 2,  5,   7,  9,
                                              ...
```

sowie die eigentliche Schnittstellenfunktion `computeInC_1`:

```
computeInC_1::
   Array (Int,Int) Int -> Array (Int,Int) Int -> Array (Int,Int) Int
computeInC_1  arr_2_b arr_2_c =
 unsafePerformPrimIO (
 (newIntArray (l_2_b,u_2_b)) `thenPrimIO` \ _barr_2_b ->
 (pack2I _barr_2_b arr_2_b)   `seqPrimIO`
 (newIntArray (l_2_c,u_2_c)) `thenPrimIO` \ _barr_2_c ->
 (pack2I _barr_2_c arr_2_c)   `seqPrimIO`
 (newIntArray ((0,0),(3,3))) `thenPrimIO` \result ->
  (_ccall_ c_function_1
          _barr_2_b (projectIdx2 2 u_2_b) (projectIdx2 1 u_2_b)
          _barr_2_c (projectIdx2 2 u_2_c) (projectIdx2 1 u_2_c)
          result (projectIdx2 2 u_2_result) (projectIdx2 1 u_2_result)
```

```
          0 ) `seqPrimIO`
 (returnPrimIO (listArray ((0,0),(3,3))(unpack2I result ((0,0),(3,3)))))
 where (l_2_b,u_2_b) = (bounds arr_2_b)
       (l_2_c,u_2_c) = (bounds arr_2_c)
       (l_2_result,u_2_result) = ((0,0),(3,3))
```

Die unübersichtliche Monadensyntax von Haskell 1.2 wurde im neuen Standard 1.3 [PHe96] deutlich aufgeräumt, was auch den obige Programmtext besser lesbar machen würde. Auf der C-Seite muß die Schnittstelle generiert werden, die vom Haskell-Programm zur Laufzeit die Daten entgegennimmt und die Schleife berechnet. Hierbei handelt es sich um die C-Funktion `c_function_1`, die oben als erster Parameter der `_ccall_`-Funktion zu sehen ist. Die C-Schnittstelle wird zusammen mit einigen Hilsfunktionen in der Datei <*source*>`_cintfc.c` abgelegt, die Köpfe der generierten Funktionen werden zur Typüberprüfung bei der Übersetzung des Haskell-Programms benötigt und finden sich daher zusätzlich in <*source*>`_intfc.h`. Der generierte C-Code für die Schnittstelle sieht wie folgt aus:

```
int c_function_1(StgByteArray _barr_2_b, long u_2_b_2, long u_2_b_1,
                 StgByteArray _barr_2_c, long u_2_c_2, long u_2_c_1,
                 StgByteArray _b_result, long u_result_2, long u_result_1,
                 long _null ){

/* allocate mem for C arrays and copy input into them */
long **b = init_2d_array( _barr_2_b, u_2_b_2, u_2_b_1);
long **c = init_2d_array( _barr_2_c, u_2_c_2, u_2_c_1);
long **_result = alloc_2d_array( u_result_2, u_result_1, _null);

long i,j,k;

/* C loop nest for LooPo: */
for(i = 0; i<=3; i++) {
  for(j = 0; j<=3; j++) {
    for(k = 0; k<=3; k++) {
      _result[i][j] = _result[i][j] + b[i][k] * c[k][j];
    }
  }
}
long2dToByteArray( _b_result, _result, u_result_2, u_result_1);

/*return bogus val*/
return 0;
}
```

Die neuen, generierten Dateien sind ohne weiteres sofort übersetzbar und berechnen die *array comprehension* sequentiell in C.

An dieser Stelle ist es angebracht, einen Blick hinter die Kulissen des Codegenerators zu werfen. Zuerst erkennt der Parser die einzelnen Parameter, die im String enthalten sind. Dabei werden die Ausdrücke schon gleich in den Typ `Exp` umgewandelt. Es ergibt sich also als Zwischenergebnis eine Parameterliste vom Typ `AA` (siehe Kapitel 7). Der Ausdruck

```
(alparser . scanner)"(+) 0 ((0,0),(3,3))
    [(i,j) := b!(i,k) * c!(k,j) | i <-[0..3], j<-[0..3], k<-[0..3]]"
```

wird zu

```
  AA (:+:) (I 0) (Ix2 (IConst 0) (IConst 0),Ix2 (IConst 3) (IConst 3))
     (AL (Ix2 (Contents "i") (Contents "j"))
         (ArrContents "b" (Ix2 (Contents "i") (Contents "k")) :*:
          ArrContents "c" (Ix2 (Contents "k") (Contents "j")))
         [LH "i" (IConst 0) (IConst 3),
          LH "j" (IConst 0) (IConst 3),
          LH "k" (IConst 0) (IConst 3)])
```

ausgewertet. In der Parameterliste werden Zahlen mit Dezimalpunkt als Fließkommazahl betrachtet, alle anderen als Integer. Die Funktion `aaTransform` wandelt den obigen Ausdruck ein einen Schleifensatz um. Zum Beispiel liefert

```
 aaTransform ((alparser . scanner)"(+) 0 ((0,0),(3,3))
     [(i,j) := b!(i,k) * c!(k,j) | i <-[0..3], j<-[0..3], k<-[0..3]]")
```

den Schleifensatz

```
For "i" (IConst 0) (IConst 3)
  (For "j" (IConst 0) (IConst 3)
     (For "k" (IConst 0) (IConst 3)
        (ArrAssign "_result" (Ix2 (Contents "i") (Contents "j"))
            (ArrContents "_result" (Ix2 (Contents "i") (Contents "j")) :+:
            (ArrContents "b" (Ix2 (Contents "i") (Contents "k")) :*:
             ArrContents "c" (Ix2 (Contents "k") (Contents "j")))))))
```

Soweit zur Betrachtung der internen Zwischenergebnisse. Da LooPo den Code zur Ansteuerung eines Parallelrechners derzeit noch nicht automatisch generieren kann, muß die Parallelisierung des Schleifensatzes zu Demonstrationszwecken manuell erfolgen.

Obiges C-Programm wurde dann mit LooPo (V1.02) [Loo] parallelisiert. Dabei kamen Scheduler und Allocator nach der Methode von Paul Feautrier zum Einsatz [Wie95]. Als Zielsprache wurde paralleles C gewählt.

```
/* LooPo program */
/*       C       */

for(t1=ceil(0);t1<=floor(3);t1+=1)
 {
  parfor(p1=ceil(0);p1<=floor(3);p1+=1)
   {
    parfor(p2=ceil(0);p2<=floor(3);p2+=1)
     {
      _result[p1,p2]=_result[p1,p2]+b[p1,t1]*c[t1,p2];
     }
   }
 }


/*    end of     */
/* LooPo program */
```

Die Existenz der Funktionen `ceil` und `floor` mit ganzzahligen Parametern ist auf die in der LooPo-Version 1.02 noch fehlende Optimierung zurückzuführen. Zu beachten ist auch, daß LooPo Variablennamen, die mit einem _ beginnen, noch nicht verarbeiten kann. Der Name `_result` wird verwendet, da er Namenskonflikte mit den übrigen Variablen vermeiden soll. Im vorliegenden Beispiel wurde er für die Bearbeitung durch LooPo temporär umbenannt.

Für den Fall, daß der Benutzer das Schleifenproramm nicht in C berechnen, sondern in Haskell interpretieren lassen möchte, fügt er folgenden Aufruf von `accumarray` ein:

```
  accumarray "(+) 0 ((0,0),(3,3)) [(i,j) := b!(i,k) * c!(k,j)
                                   |i <-[0..3], j<-[0..3], k<-[0..3]]"
             (Sto [("b",(A2 b)),("c",(A2 c))])
        where b = listArray ((0,0),(3,3)) [I    2, I    5, I    7, I    9,
                                           I (-5), I   10, I    7, I    3,
                                           I    3, I    8, I   23, I (-4),
                                           I   65, I   11, I   15, I    1]
              c = listArray ((0,0),(3,3)) [I   23, I    8, I    1, I    3,
                                           I   17, I    2, I    9, I    6,
                                           I (-9), I    2, I (-5), I   12,
                                           I    1, I (-3), I    4, I    7]
```

Kapitel 12

Zusammenfassung

Das Ziel der vorliegenden Arbeit ist die Verbindung der automatischen Parallelisierung unter Verwendung des Polytopenmodells mit der funktionalen Programmierung. Dies geschieht zu dem Zweck, den Bemühungen um die Parallelisierung von funktionalen Programmen einen neuen Aspekt hinzuzufügen.

Das Polytopenmodell wird in der automatischen Parallelisierung verwendet, um einen sequentiellen Schleifensatz, der gewissen Einschränkungen unterliegt, unter Verwendung einer Raum-Zeit-Abbildung und Veränderung der Schleifenindizes in einen parallelen Schleifensatz umzuwandeln.

Um das Polytopenmodell auf funktionale Sprachen wie Haskell anwenden zu können, müssen dort schleifenartige Strukturen identifiziert werden. Aus mehreren Alternativen wurden die *array comprehensions* ausgewählt. Da aber auch im Modell nicht beliebige Schleifensätze bearbeitet werden können, gilt das gleiche auch für die entsprechenden Haskell-Ausdrücke. Die Einschränkungen, denen diese Ausdrücke unterliegen müssen, werden in Kapitel 6 erläutert.

Existierender Haskell-Code wird einer Transformation unterzogen, bei der die auftretenden *array comprehensions* in eine imperative Schleifensprache transformiert, in die Sprache C übersetzt und dort optional parallelisiert werden. Der C-Code wird in eine Wrapper-Funktion eingebettet, die den Datenaustausch mit Haskell regelt. Im Haskell-Programm selbst wird die *array comprehension* durch den Aufruf einer anderen Haskell-Funktion ersetzt, die bezüglich des Datenaustausches das Pendant zum C-Wrapper darstellt. Der gesamte Transformationsvorgang kann mit Ausnahme der eigentlichen Parallelisierung des C-Fragmentes (LooPo ist zur Zeit noch ein interaktiv zu benutzendes Programm) automatisch ablaufen. Im Rahmen dieser Diplomarbeit wurde der Parallelisierungsschritt sowie die Anbindung an einen Parallelrechner nicht durchgeführt. Die im Anhang befindlichen Programme stellen daher die automatische Transformation mit sequentiellem Schleifensatz dar.

Möglich wird dieser Vorgang durch die Verwendung des Glasgower Haskell-Compilers, der Haskell-Programme als Zwischenschritt in C-Programme übersetzt. Nach der Übersetzung kann die nach C ausgelagerte Berechnung der *array comprehension* wieder eingefügt werden. Die Programmteile werden zu einer ausführbaren Binärdatei zusammengefügt, die auf einem sequentiellen Rechner abgearbeitet werden kann. Damit ist in dieser Arbeit theoretisch die Möglichkeit der Auslagerung von Berechnungen funktionaler Programme auf Parallelrechner geschaffen worden, auf der praktischen Seite wurde die zur Auslagerung notwendige Schnittstelle implementiert.

Für die Zukunft bleibt damit die in Kapitel 10 angesprochene Parallelrechneranbindung offen. Dabei soll in der C-Funktion die sequentielle Schleife durch einen Aufruf an den Parallelrechner ersetzt werden. Das auf dem Parallelrechner ablaufende, parallelisierte Programm sollte in diesem Rahmen auch automatisch erzeugt werden. Mit einer solchen Erweiterung dieser Diplomarbeit wäre die Möglichkeit geschaffen, zeitkritische Berechnungen eines Haskell-Programmes auf einen Parallelrechner auszulagern, während das Hauptprogramm weiterhin auf einer normalen Workstation abläuft.

Grundsätzlich gibt es für eine weitere Arbeit mehrere Entwicklungsrichtungen, die den hier verfolgten Ansatz erweitern könnten. Zum einen könnte die Transformation auf andere, schleifenartige Strukturen (wie z.B. `map`, `fold` und allgemeinere Rekursionsschemata) ausgedehnt werden. Solche Erweiterungen hätten aber mit zusätzlichen Problemen zu kämpfen, wie sie schon in Kapitel 6 angedeutet wurden.

Zum anderen ist es denkbar, die Parallelisierung von funktionalen Ausdrücken von der Mächtigkeit des Polytopenmodells her zu motivieren. Die soeben angesprochenen Funktionen können ebenso wie `accumArray` nicht in ihrer vollen Allgemeinheit bearbeitet werden. Um das Modell sinnvoll auszunutzen, könnte man eine Funktion definieren, die möglichst genau die Fähigkeiten des Modells wiederspiegelt. Ein Benutzer hätte dann die Wahl, herkömmliche Funktionen oder diese speziell optimierte Funktion zu benutzen. Dabei ist aber offen, ob eine derart direkte Abbildung des Modells in eine Funktion überhaupt möglich und sinnvoll ist.

Desweiteren sollte der Gebrauch von Parallelität vom Benutzer steuerbar sein, indem er im Einzelfall Ausdrücke von der Parallelisierung ausnehmen kann. Der Einsatz der Parallelität ist dann eher explizit anzusehen, da man sich bewußt für die Optimierung eines Ausdruckes entscheidet und bei der Programmierung die spätere Parallelisierung schon berücksichtigt. Dies folgt der herrschenden Meinung, daß der früher propagierte *dusty deck* Ansatz weit hinter dem Optimum zurückbleibt und eine Interaktion des Programmierers sinnvoll ist.

Anhang A

Der Quellcode für Scanner/Parser

Das folgende ist eine Eingabedatei für happy-0.9.

```
Association List Parser
(c) 1996 Nils Ellmenreich

derived from an introductory example of happy V0.9 parser generator
for the Haskell language

The module header
> {
> module ALparser(alparser, scanner, Var(..), Val(..), Index(..), Exp(..),
>                 LoopHead(..), AssocList(..), AA(..), Token(..) ) where

> infixl 5 :+:
> infixl 5 :@: -- minus operator. argghh.
> infixl 5 :>:
> infixl 5 :<=:
> infixl 5 :*:

> }

> %name alparser
> %tokentype { Token }

Now we declare all the possible tokens:

> %token
>       int             { TokenInt $$ }
>       double          { TokenDouble $$ }
```

```
>       var             { TokenVar $$ }
>       '='             { Equal }
>       '+'             { Plus }
>       '-'             { Minus }
>       '*'             { Times }
>       '/'             { Div }
>       '('             { OBracket }
>       ')'             { CBracket }
>       '['             { OSqBk }
>       ']'             { CSqBk }
>       '|'             { Bar }
>       ','             { Comma }
>       '>'             { Greater }
>       "<="            { LEq }
>       '!'             { Bang }
>       "<-"            { ComesFrom }
>       ":="            { AssocOp }
>       ".."            { DotDot }


> %%
```

Now we have the production rules.

```
> AccumArray :: { AA }
> AccumArray : '(' Op ')' Val '(' Idx ',' Idx ')' AList {AA $2 $4 ($6,$8) $10}

> Op :: { Exp -> Exp -> Exp }
> Op : '+'       { :+: }
>    | '-'       { :@: }
>    | '*'       { :*: }

> Val :: { Val }
> Val  : int                  { I $1 }
>      | double               { D $1 }


> Number :: { Exp }
> Number  : int               { IConst $1 }
>         | double            { DConst $1 }

> AList :: { AssocList }
> AList : '[' Idx ":=" Expr '|' GenList ']'  { AL $2 $4 $6 }
```

Generators, which turn out to be the loop heads

```
> GenList :: { [LoopHead] }
> GenList : Gen ',' GenList       { $1 : $3 }
>         | Gen                   { [ $1 ] }

> Gen : var "<-" '[' Expr ".." Expr ']'  {  LH $1 $4 $6 }

Expr is the name for expression productions rules , Exp is the Haskell
type of the corresponding algebraic datatype

> Expr :: { Exp }
> Expr : Expr '+' Term          { $1 :+: $3 }
>      | Expr '-' Term          { $1 :@: $3 }
>      | Expr '>' Term          { $1 :>: $3 }
>      | Expr "<=" Term         { $1 :<=: $3 }
>      | Term                   { ($1) }

> Term : Term '*' Factor        { ($1 :*: $3) }
>      | Factor                 { $1 }

> Factor : Number               { $1 }
>        | '(' Expr ')'         { ( $2 ) }
>        | var '!' Idx          { (ArrContents $1 $3) }
>        | var                  { Contents $1 }

> Idx :: { Index }
> Idx : '(' Expr ')'            { (Ix1 $2) }
>     | '(' Expr ',' Expr ')'   { (Ix2 $2 $4) }


> {

All parsers must declare this function,
which is called when an error is detected.
Note that currently we do no error recovery.

> happyError :: Int -> [Token] -> a
> happyError i _ = error ("Parse error in line " ++ show i ++ "\n")

Now we declare the datastructure that we are parsing.

> type Var = String

> data Val = I Int
>          | D Double deriving (Eq, Ord, Text)
```

```
> data Index = Ix1 Exp
>            | Ix2 Exp Exp deriving (Text)

> data Exp =
>    IConst Int
>  | DConst Double
>  | Contents Var
>  | ArrContents Var Index
>  | Exp :+: Exp
>  | Exp :@: Exp
>  | Exp :>: Exp
>  | Exp :<=: Exp
>  | Exp :*: Exp deriving (Text)

> data LoopHead = LH Var Exp Exp

> data AssocList = AL Index Exp [LoopHead]

> data AA = AA (Exp -> Exp -> Exp) Val (Index,Index) AssocList
```

The datastructure for the tokens...

```
> data Token
>       = TokenInt Int
>       | TokenDouble Double
>       | TokenVar String
>       | Equal
>       | Plus
>       | Minus
>       | Times
>       | Div
>       | OBracket
>       | CBracket
>       | OSqBk
>       | CSqBk
>       | Bar
>       | Comma
>       | Greater
>       | LEq
>       | Bang
>       | ComesFrom
>       | AssocOp
>       | DotDot
```

```
.. and a simple scanner that returns this datastructure.

> scanner :: String -> [Token]
> scanner [] = []
> scanner (c:cs)
>       | isSpace c = scanner cs
>       | isAlpha c = lexVar (c:cs)
>       | isDigit c = lexNum (c:cs)
> scanner ('=':cs) = Equal : scanner cs
> scanner ('+':cs) = Plus : scanner cs
> scanner ('-':cs) = Minus : scanner cs
> scanner ('*':cs) = Times : scanner cs
> scanner ('/':cs) = Div : scanner cs
> scanner ('(':cs) = OBracket : scanner cs
> scanner (')':cs) = CBracket : scanner cs
> scanner ('[':cs) = OSqBk : scanner cs
> scanner (']':cs) = CSqBk : scanner cs
> scanner (',':cs) = Comma : scanner cs
> scanner ('|':cs) = Bar : scanner cs
> scanner ('!':cs) = Bang : scanner cs
> scanner ('>':cs) = Greater : scanner cs
> scanner ('.':'.':cs) = (DotDot : scanner cs)
> scanner (':':'=':cs) = (AssocOp : scanner cs)
> scanner ('<':'-':cs) = (ComesFrom : scanner cs)
> scanner ('<':'=':cs) = (LEq : scanner cs)

 lexNum cs = TokenInt (read num) : scanner rest
        where (num,rest) = span isDigit cs

> lexNum cs = if (((head rest) == '.') && (head(tail rest) /= '.')) then
>               TokenDouble (read (num++"."++frac)) : scanner rest2
>            else
>               TokenInt (read num) : scanner rest
>          where (num,rest) = span isDigit cs
>                (frac, rest2)  = span isDigit (tail rest)

> lexVar cs =
>    case span isAlpha cs of
>       (var,rest)   -> TokenVar var : scanner rest
}
```

Anhang B

Der Quellcode des Moduls AccumArray

```
-- AccumArray.hs
--
-- (C) 1996 Nils Ellmenreich.


module AccumArray(accumarray, Agg(..), Store(..), Cmd(..), aaTransform,
                  c_expout, c) where

import ALparser

---------------------------------------------------------------------------
-- data type definitions
---------------------------------------------------------------------------
infixl 5 :>>

data Agg = A0 Val
         | A1 (Array Int Val)
         | A2 (Array (Int,Int) Val) deriving (Text)

data Store = Sto [(Var, Agg)]

-- already defined in ALparser.ly
{-

data Index = Ix1 Exp
           | Ix2 Exp Exp deriving (Text)

data Val = I Int
         | D Double deriving (Eq, Ord, Text)
```

```
type Var = String

infixl 5 :+:
infixl 5 :@: -- minus operator. argghh.
infixl 5 :>:
infixl 5 :<=:
infixl 5 :*:

data Exp =
    IConst Int
  | DConst Double
  | Contents Var
  | ArrContents Var Index
  | Exp :+: Exp
  | Exp :@: Exp
  | Exp :>: Exp
  | Exp :<=: Exp
  | Exp :*: Exp deriving (Text)

data LoopHead = LH Var Exp Exp
data AssocList = AL Index Exp [LoopHead]
-}



data Cmd =
    Nop
  | Cmd :>> Cmd
  | If Exp Cmd Cmd
  | While Exp Cmd
  | For Var Exp Exp Cmd
  | Assign Var Exp
  | ArrAssign Var Index Exp
  | DefArray Var Index Val
  | InitArray Var Index [Val]


-------------------------------------------------------------------------
-- The store for Val vars and arrays
-------------------------------------------------------------------------

-- the empty store
store0 :: Store
store0  = Sto []
```

```
-------------------------------------------------------------------------
-- getting stuff from store

-- looks for a variable in Store and returns the corresponding aggregate
lookup:: Var -> Store -> Agg
lookup v (Sto ((s,a):xs)) | v == s    = a
                          | otherwise = lookup v (Sto xs)

-- fetches the value of array at position index
ar_fetch:: Var -> Index -> Store -> Val
ar_fetch v i s = ar_getindex s i (lookup v s)

-- fetches the value from integer variable from store
fetch:: Var -> Store -> Val
fetch v s = a
          where (A0 a) = lookup v s

--------------------------------------------------------------------------
-- updating the store

-- updates the value of array at position index with value in store
ar_update:: Var -> Index -> Val -> Store -> Store
ar_update var index val s
    = (update  var (ar_putindex s index val (lookup var s)) s)

-- updates the variable/aggregate array pair in store
update::  Var -> Agg -> Store -> Store
update  var arr (Sto ((s,a):xs))
    | s == var  = (Sto ((var,arr):xs))
    | otherwise = (Sto ((s,a):st))
                where (Sto st) = (update var arr (Sto xs))
update  var arr (Sto []) =  (Sto [(var,arr)])

--------------------------------------------------------------------------
-- getting and putting of Agg's from and to store
-- case analysis on indices is necessary

-- gets the value of aggregate array at position index
ar_getindex:: Store -> Index -> Agg -> Val
ar_getindex s (Ix1 exp) (A1 arr)
    = arr!e
    where (I e) =(ensureInt exp s)
ar_getindex s (Ix2 exp1 exp2) (A2 arr)
    = arr!(e1,e2)
```

```
    where (I e1) = (ensureInt exp1 s)
          (I e2) = (ensureInt exp2 s)
-- ... add more dimensions

-- inserts new value into aggregate array at position index
ar_putindex:: Store -> Index -> Val -> Agg -> Agg
ar_putindex s (Ix1 exp) val (A1 arr)
    = A1 (arr//[ e := val ])
    where (I e) = (ensureInt exp s)
ar_putindex s (Ix2 exp1 exp2) val (A2 arr)
    = A2 (arr//[ (e1,e2) := val ])
    where (I e1) = (ensureInt exp1 s)
          (I e2) = (ensureInt exp2 s)
-- ... add more dimensions

-----------------------------------------------------------------------
-- create and initialize a new array

initarray:: Var -> Index -> [Val] -> Store  -> Store
initarray v (Ix1 exp) alist (Sto xs) = (Sto (a:xs))
                      where a = (v, A1 (listArray (0,idx) alist))
                            (I idx) = (ensureInt exp (Sto xs))

initarray v (Ix2 exp1 exp2) alist (Sto xs) = (Sto (a:xs))
                      where a = (v, A2 (listArray ((0,0),(idx1,idx2)) alist))
                            (I idx1) = (ensureInt exp1 (Sto xs))
                            (I idx2) = (ensureInt exp2 (Sto xs))
-- ... add more dimensions

-----------------------------------------------------------------------
-- just a test
stotest = ar_fetch "B" (Ix1 (IConst 3))
           (initarray "B" (Ix1 (IConst 4)) [D 4,D 5,D 8,D 7,D 9] store0)

-------------------------------------------------------------------------
-- Evaluation of loop language Exps
-------------------------------------------------------------------------

-- Evaluate Exps
evalexp:: Exp -> Store -> Val
evalexp (IConst n) s         = I n
evalexp (DConst n) s         = D n
evalexp (Contents v) s       = fetch v s
evalexp (ArrContents v idx) s = ar_fetch v idx s
```

```
evalexp (exp1 :@: exp2) s
    | (isint e1) && (isint e2)     = I (iexp1 - iexp2)
    | (isdouble e1) && (isdouble e2) = D (dexp1 - dexp2)
    | otherwise                    = error "Minus: op types not equal"
        where e1 = (evalexp exp1 s)
              e2 = (evalexp exp2 s)
              (I iexp1) = e1
              (I iexp2) = e2
              (D dexp1) = e1
              (D dexp2) = e2

evalexp (exp1 :+: exp2) s
    | (isint e1) && (isint e2)     = I (iexp1 + iexp2)
    | (isdouble e1) && (isdouble e2) = D (dexp1 + dexp2)
    | otherwise                    = error ("Plus: op types not equal.")
        where e1 = (evalexp exp1 s)
              e2 = (evalexp exp2 s)
              (I iexp1) = e1
              (I iexp2) = e2
              (D dexp1) = e1
              (D dexp2) = e2

evalexp (exp1 :*: exp2) s
    | (isint e1) && (isint e2)     = I (iexp1 * iexp2)
    | (isdouble e1) && (isdouble e2) = D (dexp1 * dexp2)
    | otherwise                    = error "Times: op types not equal"
        where e1 = (evalexp exp1 s)
              e2 = (evalexp exp2 s)
              (I iexp1) = e1
              (I iexp2) = e2
              (D dexp1) = e1
              (D dexp2) = e2

evalexp (exp1 :>: exp2) s
  | (isint e1) && (isint e2)     =  I (if (iexp1 > iexp2) then 1 else 0)
  | (isdouble e1) && (isdouble e2) =  I (if (dexp1 > dexp2) then 1 else 0)
  | otherwise                    = error "Greater: op types not equal"
    where e1 = (evalexp exp1 s)
          e2 = (evalexp exp2 s)
          (I iexp1) = e1
          (I iexp2) = e2
          (D dexp1) = e1
          (D dexp2) = e2

evalexp (exp1 :<=: exp2) s
```

```
  | (isint e1) && (isint e2)      =  I (if (iexp1 <= iexp2) then 1 else 0)
  | (isdouble e1) && (isdouble e2) =  I (if (dexp1 <= dexp2) then 1 else 0)
  | otherwise                     = error "Greater: op types not equal"
    where e1 = (evalexp exp1 s)
          e2 = (evalexp exp2 s)
          (I iexp1) = e1
          (I iexp2) = e2
          (D dexp1) = e1
          (D dexp2) = e2

-- test and assert functions for types
ensureInt:: Exp -> Store -> Val
ensureInt e s | (isint exp) = exp
              | otherwise = error "ensureInt assertion failed."
    where exp = (evalexp e s)

isint:: Val -> Bool
isint (I e) = True
isint _ = False

isdouble:: Val -> Bool
isdouble (D e) = True
isdouble _ = False
-------------------------------------------------------------------------
-- interpreter for loop language
-------------------------------------------------------------------------

interpret :: Cmd -> Store -> Store

interpret (Nop) s = s

interpret (c1 :>> c2) s          = interpret c2 (interpret c1 s)

interpret (If e c1 c2) s
    = interpret (if (evalexp e s) == (I 1) then c1
                                           else c2) s

interpret (While e c) s
    = interpret (if (evalexp e s) == (I 1) then (c :>> (While e c))
                                           else (Nop              )) s

interpret (For v e1 e2 c) s
    = interpret ((Assign v e1) :>>
                 (While ((Contents v) :<=: e2)
                     (c :>>
```

```
                    (Assign v ((Contents v) :+: (IConst 1))
                 )))) s

interpret (Assign v e) s           = update  v (A0 (evalexp e s)) s

interpret (DefArray v idx nul) s    = initarray v idx (repeat nul) s
interpret (InitArray v idx alist) s = initarray v idx alist s

interpret (ArrAssign v idx e) s     = ar_update v idx a s
    where a = evalexp e s

-- Shortcut
program :: Cmd -> Store
program a = interpret a store0

-------------------------------------------------------------------------
-- the accumArray transformer
-------------------------------------------------------------------------

-- THE function - takes all parameters like "accumArray" in a String, and needs
-- all used variables arrays in a store. It will the compute the result
-- in Haskell. Note that the returned value, unlike accumArray, is of type
-- Agg rather than an array

accumarray:: String -> Store -> Agg
accumarray acalist store =
    lookup "A" (interpret (DefArray "A" upidx nullelem :>>
                aaTransform (AA op nullelem (lowidx,upidx) alist)) store)
    where (AA op nullelem (lowidx,upidx) alist) =((alparser . scanner) acalist)

-- takes the internal representation of an accumulation array
-- and transforms it into an imperative loop program
aaTransform :: AA -> Cmd
aaTransform (AA op nullelem (lowidx,upidx) (AL idxExp rexp lhlist)) =
     (generateLoopNest lhlist
         (ArrAssign "A" idxExp
             (op (ArrContents "A" idxExp) rexp)))

-- generates a loop nest according to lhlist around the loop body
generateLoopNest:: [LoopHead] -> Cmd -> Cmd
generateLoopNest [] lbody = lbody
generateLoopNest (lh:lhlist) lbody = (For var lowexpr upexpr
                                  (generateLoopNest lhlist lbody))
                           where (LH var lowexpr upexpr) = lh
```

```
-- assocs?
-- returns the assoc list (= contents) of an array
assocs1:: Var -> Store -> [Assoc Int Val]
assocs1 k (Sto ((s, arr):xs))
    | s == k = assocs t
    | s /= k = assocs1 k  (Sto xs)
            where (A1 t) = arr

assocs2:: Var -> Store -> [Assoc (Int,Int) Val]
assocs2 k (Sto ((s, arr):xs))
    | s == k = assocs t
    | s /= k = assocs2 k  (Sto xs)
            where (A2 t) = arr

----------------------------------------------------------------------------
-- C output

c :: Cmd -> String
c prog = "\n/* C loop nest for LooPo: */\n" ++(c_out 0 prog)

c_out :: Int -> Cmd -> String
c_out n (left :>> right) = (c_out n left) ++ (c_out n right)
c_out n (DefArray rest1 rest2 rest3) = ""
c_out n (InitArray rest1 rest2 rest3) = ""
c_out n (Nop) = ""
c_out n (Assign v e) = (spaces n) ++ v ++ " = " ++ (c_expout e) ++ ";\n"
c_out n (ArrAssign v idx e) =
    (spaces n) ++ v ++ (c_showidx idx) ++ " = " ++ (c_expout e) ++ ";\n"
c_out n (While e c) =
    (spaces n) ++ "while (" ++ (c_expout e) ++ ") {\n"
    ++ (c_out (n+2) c)
    ++ (spaces n) ++ "}\n"
c_out n (For v e1 e2 c) =
    (spaces n) ++ "for(" ++ v ++ " = " ++ (c_expout e1) ++ "; "
    ++ v ++ "<=" ++ (c_expout e2) ++ "; " ++ v ++ "++) {\n"
    ++ (c_out (n+2) c)
    ++ (spaces n) ++ "}\n"
c_out n (If e c1 c2) =
        (spaces n) ++ "if (" ++ (c_expout e) ++ ") { \n"
        ++ (c_out (n+2) c1)
        ++ (spaces n) ++ "}else{\n"
        ++ (c_out (n+2) c2)
        ++ (spaces n) ++ "}\n"
```

```
-- outputs an expression
c_expout:: Exp -> String
c_expout (IConst i) = (show i)
c_expout (DConst d) = (show d)
c_expout (Contents v) = v
c_expout (ArrContents v idx) = v  ++ (c_showidx idx)
c_expout ( e1 :+: e2) = (c_expout e1) ++ " + " ++ (c_expout e2)
c_expout ( e1 :@: e2) = (c_expout e1) ++ " - " ++ (c_expout e2)
c_expout ( e1 :*: e2) = (c_expout e1) ++ " * " ++ (c_expout e2)
c_expout ( e1 :>: e2) = (c_expout e1) ++ " > " ++ (c_expout e2)
c_expout ( e1 :<=: e2) = (c_expout e1) ++ " <= " ++ (c_expout e2)

-- outputs an index
c_showidx :: Index -> String
c_showidx (Ix1 e) = "[" ++ (c_expout e) ++ "]"
c_showidx (Ix2 e1 e2) = "[" ++ (c_expout e1) ++ "][" ++ (c_expout e2) ++ "]"

-------------------------------------------------------------------------------
-- AUXILIARY FUNCTIONS

-- returns string of n spaces
spaces:: Int -> String
spaces 0 = ""
spaces (n+1) = " "++(spaces n)
spaces _ = error "spaces: neg. number."

------------------------------ THE END ------------------------------------
```

Anhang C

Der Quellcode des automatischen Codegenerators

```
{-
   Nils Ellmenreich
   1996

   mk_HaskellComputeInC  - generate Haskell function to call the C
   computation
-}

module Main where

import ALparser
import AccumArray

type StoreList = [(String, Int)]     -- (varname, dim)

main :: IO ()
main = gets >>= \s ->
       (putStr ((mk_HaskellComputeInC (parsetree s))
                ++"\n----cfun----\n"
                ++(fst (mk_cfunction (parsetree s)))
                ++"\n----chead----\n"
                ++(snd (mk_cfunction (parsetree s)))
                ))
    where parsetree=(alparser . scanner)

gets = getChar >>= \c ->
       (case c of
```

```
                  ('\n') -> (return "")
                  otherwise -> (gets >>= \s -> return (c:s)))

mk_cfunction :: AA -> (String,String)
mk_cfunction (AA op nullelem idx alist) =
    (funccall
    ++"{\n\n"
    ++"/* allocate mem for C arrays and copy input into them */\n"
    ++(foldr (++) "" [(mk_initArray name dim)|(name,dim) <- storelist, dim>0])
    ++"long "++(take resDim (repeat '*'))++"A = alloc_"
    ++(show resDim)++"d_array("
    ++(tail (foldr (++) "" [", u_result_"++(show(resDim+1-pos))
                            |pos<-[1..resDim]]))
              ++", _null);\n\n"
    ++"long "++loopidxlist++";\n"
    ++(c (aaTransform (AA op nullelem idx alist)))
    ++"long"++(show resDim)++"dToByteArray( _b_result, A"
    ++(foldr (++) "" [", u_result_"++(show(resDim+1-pos))
                      |pos<-[1..resDim]])
    ++");\n\n/*return bogus val*/\nreturn 0;\n}",
    funccall++";\n")
    where  mk_cfunParam:: String -> Int -> String
           mk_cfunParam name dim =
               "StgByteArray _barr_"++id++", "
               ++(foldr (++) "" ["long u_"++id++"_"++(show(dim+1-pos))++", "
                                 |pos<-[1..dim]])
               where id = (show dim)++"_"++name
           mk_initArray:: String -> Int -> String
           mk_initArray name dim =
               "long "++(take dim (repeat '*'))++name++" = init_"
               ++(show dim)++"d_array("
               ++" _barr_"++id
               ++(foldr (++) "" [", u_"++id++"_"++(show(dim+1-pos))
                                 |pos<-[1..dim]])
               ++");\n"
               where id = (show dim)++"_"++name

           storelist = (cons_storelist valexpr [])
                       ++(foldr (++) []
                        [((cons_storelist exp1 [])++(cons_storelist exp2 []))
                         |(LH _ exp1 exp2) <- lhlist ])
           loopidxlist= tail (foldr (++) []
                             [","++var|(LH var _ _ ) <- lhlist ])
           (AL _ valexpr lhlist) = alist
           (resDim, resIdxType, resIdx) = get_resultDim idx
```

```
          funccall =
             "int c_function_X( "
             ++(foldr (++) "" [(mk_cfunParam name dim)
                              |(name,dim) <- storelist, dim>0])
             ++(foldr (++) "" [(" long "++name++",")
                              |(name,dim) <- storelist, dim==0])
             ++"StgByteArray _b_result "
             ++(foldr (++) "" [", long u_result_"++(show(resDim+1-pos))
                              |pos<-[1..resDim]])
             ++" , long _null )"

mk_HaskellComputeInC :: AA -> String
mk_HaskellComputeInC (AA op nullelem idx alist) =
    "(computeInC_X"
    ++(foldr (++) "" [" "++name|(name,dim)<-storelist])++")\n\n"
    ++(mk_FctHeader storelist resDim )
    ++ " unsafePerformPrimIO ( \n"
    ++ (foldr (++) "" [(mk_ByteArrayDef name dim)
                      | (name, dim) <- storelist, dim>0])
    ++ " (newIntArray "++resIdx++") `thenPrimIO` \\result -> \n "
    ++ " (_ccall_ c_function_X "
    ++(foldr (++) "" [(mk_ccallParam name dim)|(name,dim) <- storelist, dim>0])
    ++(foldr (++) "" [(name++" ")|(name,dim) <- storelist, dim==0])
    ++" result "++(projectOnDim "result" resDim resDim)
    ++" "++(show intnull)
    ++" ) `seqPrimIO`\n"
    ++" (returnPrimIO (listArray "++resIdx++"(unpack"++(show resDim)
    ++"I result "++resIdx++")))"
    ++")\n where "
    ++(mk_InputBounds storelist resDim resIdx)
   where (resDim, resIdxType, resIdx) = get_resultDim idx
         mk_ccallParam:: String -> Int -> String
         mk_ccallParam name dim = " _barr_"++(show dim)++"_"++name++" "
                                  ++(projectOnDim name dim dim)++" "
         projectOnDim:: String -> Int -> Int -> String
         projectOnDim name dim pos
             | (dim == 0) = ""
             | (dim == 1) = " u_"
                          ++(show dim)++"_"++name
             | otherwise = if (pos > 0) then
                              "(projectIdx"++(show dim)
                              ++" "++(show pos)++" u_"
                              ++(show dim)++"_"++name
                              ++") "
                              ++(projectOnDim name dim (pos-1))
```

```
                             else
                              ""
        storelist = (cons_storelist valexpr [])
                    ++(foldr (++) []
                       [((cons_storelist exp1 [])++(cons_storelist exp2 []))
                        |(LH _ exp1 exp2) <- lhlist ])
        (AL _ valexpr lhlist) = alist
        (I intnull) = nullelem

get_resultDim idx =
    (case idx of
        ((Ix1 l ), (Ix1 u)) -> (1, "(Int)",
                                "("++(c_expout l)++","
                                ++(c_expout u)++")")
        ((Ix2 l1 l2),(Ix2 u1 u2)) -> (2, "(Int,Int)",
                                     "(("++(c_expout l1)++","
                                     ++(c_expout l2)++"),("
                                     ++(c_expout u1)
                                     ++","++(c_expout u2)++"))" )
        otherwise -> (error "unsupported dim."))

mk_FctHeader :: StoreList -> Int -> String
mk_FctHeader store resDim  = (typeDef ++ funParam ++ " = \n")
   where typeDef = ("computeInC_X"++" :: "
                   ++(foldr (++) "" [ ((mk_TypeFromDim dim)++" -> ")
                                    |(_, dim) <- store]))
                   ++(mk_TypeFromDim resDim)++"\n"
        funParam = ("computeInC_X"++" "
                    ++(foldr (++) "" [(" arr_"++(show dim)++"_"++name)
                                     |(name, dim) <- store, (dim > 0)])
                    ++(foldr (++) "" [ (" "++name)
                                     |  (name, dim) <- store,(dim == 0)]))

mk_ByteArrayDef:: String -> Int -> String
mk_ByteArrayDef varname dim =
    " (newIntArray (l_"++id++",u_"++id
    ++")) `thenPrimIO` \\ _barr_"++id++" -> \n (pack"++(show dim)++"I "
    ++"_barr_"++id++" arr_"++id++")   `seqPrimIO`\n"
    where id = (show dim)++"_"++varname


mk_InputBounds:: StoreList -> Int -> String -> String
mk_InputBounds store resDim resIdx =
    (foldr (++) "" [ ("(l_"++(show dim)++"_"++name
                    ++",u_"++(show dim)++"_"++name
```

```
                     ++") = (bounds arr_"++(show dim)++"_"++name++")\n       ")
                    |  (name, dim) <- store,(dim > 0)])
    ++"(l_"++(show resDim)++"_result"
    ++",u_"++(show resDim)++"_result"
    ++") = "++resIdx++"\n"

mk_TypeFromDim:: Int -> String
mk_TypeFromDim dim =
    ( case dim of
          (0) -> "Int"
          (1) -> "Array Int Int"
          (2) -> "Array (Int,Int) Int")


-- Evaluate Exps
-- indices in den loopheads
cons_storelist:: Exp -> StoreList -> StoreList
cons_storelist (IConst n) s          = s
cons_storelist (DConst n) s          = s
cons_storelist (Contents v) s        = (v,0):s
cons_storelist (ArrContents v idx) s = (v,dim):s
    where dim = (case idx of
                     (Ix1 _) -> 1
                     (Ix2 _ _) -> 2)
cons_storelist (exp1 :@: exp2) s =
    (cons_storelist exp1 s)++(cons_storelist exp2 s)
cons_storelist (exp1 :+: exp2) s =
    (cons_storelist exp1 s)++(cons_storelist exp2 s)
cons_storelist (exp1 :*: exp2) s =
    (cons_storelist exp1 s)++(cons_storelist exp2 s)
cons_storelist (exp1 :>: exp2) s =
    (cons_storelist exp1 s)++(cons_storelist exp2 s)
cons_storelist (exp1 :<=: exp2) s =
    (cons_storelist exp1 s)++(cons_storelist exp2 s)
```

Anhang D

Der Quellcode von mkCinterface

```
#!/bin/bash
# (c) 1996 Nils Ellmenreich
# automatic replacement of an accumArray expression by a function call
# to conduct the computation in C rather than Haskell (possibly parallelised)

echo Replacement of accumArray expressions by ccalls to C

if [ "$#" != "1" ]; then
  echo "Wrong number of parameters!"
  echo "Usage: $0 <haskellfile>"
  exit 1
fi

SRCFILE=$1
INPUTFILE=$SRCFILE
OUTFILE=tmpout
LIBDIR=./aalib
COUTFILE=${SRCFILE%.hs}_cintfc.c
HOUTFILE=${SRCFILE%.hs}_intfc.h
RUN=1

cp $LIBDIR/c_interface.prelude $COUTFILE
cp $LIBDIR/c_header_interface.prelude $HOUTFILE

# find first occurence of an accumArray expression
# assuming all parameters to ammuArray are on the same line
awk '
/accumArray/ {
  n = index($0,"accumArray")+10
```

```
  print substr($0,n,length($0)-n+1)
  exit}
' $INPUTFILE > alistfile

while [ -s alistfile ] ; do
  echo -e "\nsubstitution  #" $RUN
  echo "Starting to replace:"
  cat alistfile
  # generate Haskell and C interface functions
  ./intfc_codegen < alistfile > haskell_c_interface

  # split 'em up
  # first, the haskell function
  awk '
  NR == 1,/^----cfun----/ {
    if ($0 != "----cfun----") {
      print
    }
  }
  ' haskell_c_interface > out$RUN.hs

  # second, the corresponding C function
  awk '
  /^----chead----/ { p=0 }
  p == 1 { print }
  /^----cfun----/ { p=1 }
  ' haskell_c_interface > out$RUN.c

  # third, the C header
  awk '
  p == 1 { print }
  /^----chead----/ { p=1 }
  ' haskell_c_interface > out$RUN.h

  # print input program up to accumArray call to output
  awk '
    !/accumArray/ { print }
    /accumArray/ {
      n = index($0,"accumArray")
      printf "%s", substr($0,0,n-1)
      exit
    }
  ' $INPUTFILE > $OUTFILE

  # add the the call to the Haskell interface function to output,
```

```
  # replacing the original accumArray statement
  awk '
  { if (NR == 1)
    { print
      exit
    }
  }
  ' out$RUN.hs >> $OUTFILE

  # append the rest of the input to the output
  awk '
  p == 1 { print }
  /accumArray/ { p = 1 }
  ' $INPUTFILE >> $OUTFILE

  # append the Haskell interface implementation to the output
  awk ' { if (NR != 1) { print } } ' out$RUN.hs >> $OUTFILE
  sed -e 's/computeInC_X/computeInC_'$RUN'/g' \
      -e 's/c_function_X/c_function_'$RUN'/g' $OUTFILE > tmpin

  # the same for C
  sed  's/c_function_X/c_function_'$RUN'/g' out$RUN.c >> $COUTFILE
  sed  's/c_function_X/c_function_'$RUN'/g' out$RUN.h >> $HOUTFILE

  # remove temps
  /bin/rm alistfile haskell_c_interface out$RUN.*
#  mv $OUTFILE tmpin
  INPUTFILE=tmpin

  # find first occurenca of an accumArray expression
  # assuming all parameters to ammuArray are on the same line
  awk '
    /accumArray/ {
    n = index($0,"accumArray")+10
    print substr($0,n,length($0)-n+1)
    exit}
  ' $INPUTFILE > alistfile
  RUN=$[ $RUN + 1 ]
done

rm tmpout alistfile
mv tmpin ${SRCFILE%.hs}_intfc.hs
cat $LIBDIR/computeInC.prelude >>  ${SRCFILE%.hs}_intfc.hs
echo Done.
```

Literaturverzeichnis

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Ber75] Klaus J. Berkling. Reduction languages for reduction machines. In *2nd International Symposium on Computer Architecture*. ACM/IEEE, 1975.

[BW87] Richard Bird and Philip Wadler. *Introduction to Funktional Programming*. Prentice Hall, 1987.

[Chu41] Alonzo Church. The calculi of lambda conversion. *Annals of Math. Studies*, No. 6, 1941. Princeton University Press.

[Col89] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[DHHW93] Jack J. Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker. A proposal for a userlevel, message passing interface in a distributed memory evironment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[DM82] Luís Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium of Programming Languages*, pages 207–212, January 1982.

[EFG+96] Nils Ellmenreich, Peter Faber, Martin Griebl, Robert Günz, Harald Keimer, Wolfgang Meisl, Sabine Wetzel, Christian Wieninger, and Alexander Wüst. *LooPo - Loop Parallelization in the Polytope Model*. Universität Passau, 1996. http://www.uni-passau.de/loopo/doc/loopo_doc/loopo_doc.html.

[Ell95] Nils Ellmenreich. Das Frontend von LooPo. Unpublished Report, 1995. http://www.uni-passau.de/loopo/doc/ellmenreich-p.ps.gz.

[Fab95] Peter Faber. LooPo Target-Output. Unpublished Report, 1995. http://www.uni-passau.de/loopo/doc/faber-p.ps.gz.

[Fab97] Peter Faber. Transformation von Shared-Memory-Programmen zu Distributed-Memory-Programmen. Diplomarbeit, Universität Passau, 1997. In preparation.

[FW76] Daniel P. Friedman and David S. Wise. The impact of applicative multiprogramming on multiprocessing. In *Proceedings of the 1976 International Conference on Parallel Processing*, 1976.

[GL96a] Martin Griebl and Christian Lengauer. The Loop Parallelizer Loopo. In Michael Gerndt, editor, *Proc. of the 6th Workshop on Compilers for Parallel Computers (CPC'96)*, volume 21/1996 of *Konferenzen des Forschungszentrums Jülich*. Forschungszentrum Jülich GmbH, 1996.

[GL96b] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo—announcement. In *Ninth Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer Verlag, 1996. To appear.

[GM96] Andy Gill and Simon Marlow. Happy 0.9 parser generator. ftp://ftp.dcs.gla.ac.uk/pub/haskell/happy, February 1996.

[Gün96] Robert Günz. The new LooPo scanner and parser. Unpublished Report, 1996. http://www.uni-passau.de/loopo/doc/guenz-p.ps.gz.

[Ham94] Kevin Hammond. Parallel functional programming: An introduction. In *1st Intl. Symp. on Parallel Symbolic Computing,* Linz, 1994.

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

[HF92] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hil92] Jonathan M. D. Hill. Data Parallel Haskell: Mixing old and new glue. Technical Report QMW-DCS-611, Department of Computer Science, Queen Mary & Westfield College, December 1992.

[HMT88] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ml, version 2. Technical Report ECS-LFCS-88-62, Laboratory for Foundation of Computer Science, Univ. of Edinburgh, 1988.

[HPJW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell, A Non-Strict Purely Functional Language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[HS86] Paul Hudak and L. Smith. Parafunctional programming: A paradigm for programming multiprocessor systems. In *ACM Symposium on the Principles of Programming Languages (POPL'86)*, pages 243–254. ACM Press, January 1986.

[Hud86] Paul Hudak. Para-functional programming. *IEEE Computer*, August 1986.

[Hud93] Paul Hudak. Mutable Abstract Datatypes (or How to have your state and munge it too). Technical Report YALEU/DCS/RR-914, Department of Computer Science, Yale University, New Haven, CT 06520, May 1993.

[JD93] Mark P. Jones and Luc Duponcheel. Composing Monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.

[JM95] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming*, Lecture Notes in Computer Science 925. Springer Verlag, May 1995.

[Jon96] Mark Jones. Hugs 1.3 — embracing functional programming. http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html, 1996.

[Kei96] Harald Keimer. Vergleich von Verfahren zur Datenabhängigkeitsanalyse. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[KMW67] Richard M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[KW92] David King and Philip Wadler. Combining monads. In *Glasgow functional programming workshop*, Springer-Verlag Workshops in Computing Series, pages 134–143, July 1992.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

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

[Loo] The polyhedral loop parallelizer LooPo. http://www.uni-passau.de/~lengauer/loopo/.

[Mac71] Saunders MacLane. *Category Theory for the Working Mathematician.* Springer Verlag, 1971.

[MAE+64] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *Lisp 1.5 Programmer's Manual.* MIT Press, 1964.

[Mei96] Wolfgang Meisl. Practical methods for scheduling and allocation in the polytope model. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[Mog90] Eugenio Moggi. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Univ. of Edinburgh, 1990.

[Pad88] Peter Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1988.

[Par93] Parsytec Computer GmbH. *PARIX 1.2 Software Documentation*, March 1993.

[PE93] Rinus Plasmeijer and Marco van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[PHe96] John Peterson and Kevin Hammond (editors). Report on the Programming Language Haskell, A Non-Strict Purely Functional Language (version 1.3). Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996.

[Pie91] Benjamin Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison Wesley, 1989.

[Sch93] Wolfgang Schreiner. Parallel functional programming — an annotated bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler Universität, Linz, May 1993.

[Ski94] David Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[THMJ+96] Phil Trinder, Kevin Hammond, Jim S. Mattson Jr, Andrew Partridge, and Simon Peyton Jones. GUM: A portable parallel implementation of Haskell. In *Proc. of ACM SIPGLAN Conf. on Programming Languages Design and Implementation (PLDI'96)*, pages 79–88. ACM Press, May 1996.

[Tho96] Simon Thompson. *Haskell – The Craft of Functional Programming.* Addison-Wesley, 1996.

[Wad90] Philip Wadler. Comprehending Monads. In *Conference on Lisp and Functional programming*, pages 61–78. ACM Press, June 1990.

[Wad92a] Philip Wadler. Monads for functional programming. In *Lecture Notes for Marktoberdorf Summer School on Program Design Calculi*, pages 233–264. Springer-Verlag, August 1992.

[Wad92b] Philip Wadler. The essence of functional programming. In *Proc. of Principles of Programming Languages (POPL'92)*, pages 1–14. ACM Press, January 1992.

[Wad95] Philip Wadler. How to declare an imperative (invited paper). In John Lloyd, editor, *International Logic Programming Symposium (ILPS'95)*, pages 18–32. MIT Press, December 1995.

[Wet95] Sabine Wetzel. Automatic code generation in the polytope model. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Wie95] Christian Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Wüs95] Alexander Wüst. DISPO2 — Graphische Darstellung der Indexräume verschachtelter Schleifen. Unpublished Report, 1995. http://www.uni-passau.de/loopo/doc/wuest-p.ps.gz.

Eidesstattliche Erklärung

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

Passau, 18. Dezember 1996

(Nils Ellmenreich)