DIPLOMARBEIT

Einführung und Optimierung von Kontextargumenten in Kontinuationsprolog

ausgeführt am Institut für Computersprachen,
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
o.Univ. Prof. Dipl.-Ing. Dr. Manfred Brockhaus
und
Univ. Ass. Dipl.-Ing. Dr. Ulrich Neumerkel
als verantwortlich mitwirkenden Universitätsassistenten

durch

Markus Schordan

Lindengasse 49/2/10
A-1070 Wien
Österreich

Wien, April 1997

Zusammenfassung

Die vorliegende Diplomarbeit behandelt die Optimierung von Prologprogrammen durch Programmtransformationen in Kontinuationsprolog. Durch das Einführen und Optimieren von Kontextargumenten wird eine verbesserte interprozedurale Registerallokation der WAM erreicht, sodaß die Effizienz von Programmen erhöht wird.

Schlüsselwörter: Programmtransformation, Prolog, Transformationsregeln, Transformationsstrategien, binäres Prolog, Kontinuationsprolog, Low-Level Prolog, WAM, BinWAM.

Abstract

This work deals with the optimization of Prolog programs using program transformations in Continuation Prolog. By introducing and optimizing context arguments the interprocedural register allocation of the WAM is improved causing a more efficient program execution.

Keywords: program transformation, Prolog, transformation rules, transformation strategies, binary Prolog, Continuation Prolog, WAM, BinWAM.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Effizienz von Programmen ist ein entscheidender Faktor für deren Anwendungsmöglichkeiten. Ein optimiertes Programm ist aber meist weniger gut lesbar als ein übersichtlich formuliertes Programm. Im Idealfall wird durch ein Programm nur beschrieben, was berechnet werden soll und dem Interpreter überlassen einen effizienten Algorithmus zu finden. In Prolog wird dieses Konzept teilweise umgesetzt. Durch die deklarative Bedeutung eines Programms ist nicht festgelegt, wie dieses auszuführen ist. Aus Effizienzgründen besitzt aber jedes Programm auch eine prozedurale Bedeutung. Diese wird durch den Lösungsalgorithmus festgelegt, den Prolog verwendet, um Anfragen zu beweisen. Will man die Effizienz des Lösungsalgorithmus erhöhen, ist es nötig, die Ausführung eines Programms prozedural nachzuvollziehen. Dies ist allerdings sehr mühsam und durch derartige Optimierungen werden Programme schlecht lesbar und schwer wartbar.

Durch Programmtransformationen werden diese beiden Gegensätze zwischen übersichtlicher und optimierter Darstellung vereint. Das Programm wird ausschließlich in Hinsicht auf Klarheit der Darstellung formuliert und durch Transformationen wird dann ein effizientes Programm erzeugt.

Es gibt verschiedene Ansätze, um Prologprogramme zu optimieren. Wir werden Transformationen kennenlernen, die die rekursive Struktur von Programmen verändern. Diese Optimierungen werden durch die Techniken Falten/Entfalten erreicht. Weiters werden wir Optimierungen in Kontinuationsprolog eingehend untersuchen. Kontinuationsprolog ist eine Zwischensprache in der Übersetzung von Prolog nach einer abstrakten Maschine und für die Formulierung von Optimierung besonders gut geeignet. Wird Prolog auf einer WAM ausgeführt, so wird die Effizienz von Programmen dadurch vermindert, daß Werte nicht über Prädikatsgrenzen hinweg im selben Register gehalten werden können. Es sind Datenbewegungen notwendig, die in prozeduralen Sprachen nicht nötig sind. Durch Kontinuationstransformationen werden wir die Registerallokation der prozeduraler Sprachen annähern können.

In Kapitel 2 wird die Programmiersprache Prolog vorgestellt. Wir gehen besonders auf jene Programmkonstrukte ein, die aufgrund ihrer häufigen Verwendung interessant für Optimierungen sind. Dies sind Akkumulatoren und Listendifferenzen. Beiden ist gemeinsam, daß Zwischenvariablen verwendet werden, um Zustände weiterzureichen. In Kapitel 3 werden wir sehen, daß einige dieser Variablen durch Entfalten und Falten von Programmen optimiert werden können. Wir werden die Auswirkungen von Programmtransformationen auf die Semantik von Prologprogrammen untersuchen und feststellen, daß das Selbstfalten von Klausen vermieden werden muß. Weiters werden wir sehen, daß durch die Veränderung der rekursiven Struktur von Programmen redundante Berechnungen beseitigt werden können. Ein weiterer Ansatz wird das Ändern der Datenrepräsentation sein. So kann etwa die Effizienz eines Programms durch das Ersetzen normaler Listen durch Listendifferenzen erhöht werden. Die Optimierung von Listendifferenzen wird dann Gegenstand des Kapitels 4 sein. Wir werden sehen, wie durch Kontinuationstransformationen Kontextvariablen und Listendifferenzen optimiert werden können. Alle Optimierungen werden in Kontinuationsprolog durchgeführt werden. Durch diesen Formalismus werden die Optimierungen übersichtlich dargestellt. Das optimierte Programm werden wir nach verschiedenen abstrakten Maschinen übersetzen. Programme werden zunächst nach binärem Prolog oder auch Low-Level Prolog übersetzt werden, um dann in Code für eine abstrakte Maschine übersetzt zu werden. In Kapitel 5 werden wir den Aufbau und die Funktionsweise der WAM kennenlernen und die Optimierungen, die durch Kontinuationstransformationen erreicht werden, in Begriffen der WAM formulieren.

Kapitel 2

Prolog

In diesem Kapitel stellen wir die Programmiersprache Prolog vor. Nach einem kurzen geschichtlichen Abriß erläutern wir die grundlegenden Begriffe von Prolog. Wir werden sehen, daß Prolog aus Effizienzgründen sowohl eine prozedurale als auch eine deklarative Semantik besitzt. Dieses Trennung war der entscheidende Schritt, um auch in der Logikprogrammierung ein respektables Laufzeitverhalten zu erreichen. Weiters werden wir einige Programmiertechniken kennenlernen. So wird etwa die Verwendung von Akkumulatoren und Listendifferenzen beschrieben. Da diese Programmkonstrukte für eine effiziente Implementierung einen hohen Stellenwert einnehmen, sind sie besonders interessant für Optimierungen. Diese werden Gegenstand der folgenden Kapitel sein.

2.1 Geschichte

Die logische Programmierung entstand aus dem Wunsch, logische Theoreme mechanisch zu beweisen. Die ersten Versuche basierten auf Robinsons 1965 veröffentlichten Unifikationsalgorithmus und Resolutionsverfahren [Rob65]. Diese Ansätze als Grundlage für Berechnungsmechanismen zu verwenden, brachte aber nur sehr langsame Implementierungen hervor. Ein wichtiger Schritt, die Effizienz von Logikprogrammen zu erhöhen, wurde von Kowalski und Colmerauer getan. Kowalski formulierte die prozedurale Interpretation von Horn-Klause-Logik. Er zeigte, daß ein Axiom

$$K \textit{ wenn } Z_1 \textit{ und } Z_2 \textit{ und } \ldots \textit{ und } Z_n.$$

wie eine Prozedur einer rekursiven Programmiersprache gelesen und ausgeführt werden kann [Kow79b]. Das Konzept einer logischen Programmiersprache besteht darin, das für die Lösung eines Problems nötige Wissen sowie die Problemspezifikation in deklarativer Form anzugeben. Die entsprechende deklarative Lesart für obiges Axiom ist: „K ist wahr, wenn alle Ziele Z_i wahr sind“, mit $1 \leq i \leq n$. Das Auffinden eines Lösungsalgorithmus ist dann Aufgabe des Interpreters. Ausschlaggebend für die Effizienz einer logischen Programmiersprache ist die Auswahlreihenfolge der einzelnen Klausen und wie die Konjunktionen von Literalen im Rumpf einer Klause ausgeführt werden. Der entscheidende Schritt war, daß Prolog aus Effizienzgründen das deklarative Konzept nur partiell realisiert. Colmerauer wählte eine Selektionsfunktion mit Tiefensuche. Literale werden von links nach rechts abgearbeitet. Dieser Ansatz ermöglichte eine Implementierung, die Prozeduraufrufe ähnlich wie in prozeduralen Programmiersprachen realisiert. Colmerauer stellte sein System 1972 vor. Es hieß Prolog (*programmation en logique*).

2.2 Grundbegriffe

Ein Prologprogramm ist eine Menge von Klausen. Die allgemeine Form einer Klause ist:

$$Kopf \leftarrow Rumpf.$$

Eine Prolog-Klause genau dieser Form wird als Regel bezeichnet. Eine Klause ohne Rumpf heißt Faktum. Eine Klause heißt Anfrage, wenn der Kopf fehlt. Sie besteht nur aus dem Rumpf und hat die Form

$$\leftarrow Rumpf.$$

Der Klausekopf wird durch eine Atomformel, der Klauserumpf durch eine Folge von Atomformeln beschrieben. In Klausen werden Atomformeln als Literale bezeichnet. Atome sind Konstanten, Funktionsnamen und Prädikatsnamen. Terme werden wie üblich induktiv definiert, wobei Konstanten und Zahlen nullstellige Terme sind. Ein Term besteht aus einem Funktor und einer bestimmten Anzahl von Argumenten. Die Anzahl der Argumente heißt Arität des Terms. Ein Term hat die Form $f(a_1, a_2, \ldots, a_n)$, auch als f/n bezeichnet, mit Arität n. Ein Term mit Arität 0 heißt Konstante. Die Argumente eines Terms können ihrerseits wieder Terme sein. Listen sind zweistellige Funktionssymbole und besitzen auch eine interne Termstruktur mit dem Punkt '.' als Funktionszeichen.

2.3 Ausführungsmodell und grundlegende Konstrukte

Prolog besitzt sowohl eine deklarative als auch eine prozedurale Semantik. Ein Prologprogramm ist eine Menge von Prädikaten bzw. Prozeduren. Jene Klausen, deren Köpfe den gleichen Namen und die gleiche Arität aufweisen, definieren vom deklarativen Standpunkt aus ein Prädikat. Unter dem Aspekt der prozeduralen Semantik bilden solche Klausen eine Prozedur.

Ein Prozeduraufruf erfolgt in Prolog durch Unifikation der aktuellen Parameter. Die Unifikation entspricht der Argumentübergabe und der Zuweisung in imperativen Programmiersprachen. Im Folgenden wird zuerst die Unifikation von zwei Termen definiert. Dann wird auf das Ausführungsmodell von Prolog eingegangen.

2.3.1 Unifikation

Um zwei Terme unifizieren zu können, müssen Variablen durch Terme ersetzt werden. Die Ersetzung einer Variable durch einen Term heißt Substitution.

Definition. Eine Substitution θ ist eine Abbildung, die einer endlichen Anzahl Variablen $v_1 \ldots v_n$ (evtl. variablenbehaftete) Terme $t_1 \ldots t_n$ zuordnet. Wir schreiben dies als: $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$. Die Anwendung einer Substitution θ auf einen Term T wird postfix als Tθ notiert [Yas95].

Definition. Zwei Terme t und u sind *unifizierbar*, wenn eine Substitution θ existiert, die den Variablen von t und u Terme zuordnet, sodaß $t\theta$ und $u\theta$ identisch sind [GH89].

D.h., t und u sind unifizierbar, wenn die Variablen in t und u durch solche Terme ersetzt werden können, daß diese Terme nach dieser Ersetzung syntaktisch übereinstimmen. Eine notwendige Bedingung für die Unifizierbarkeit von zwei Termen ist daher, daß die Funktoren von t und u gleiche Arität haben und deren Namen übereinstimmen.

Beispiele:

- f(X,a) und f(b,Y) sind unifizierbar mit der Substitution θ={X/b,Y/a}. Es wird also X durch b und Y durch a ersetzt.
- f(X,a) und g(b,a) sind nicht unifizierbar, da die Namen der Funktoren nicht übereinstimmen.
- f(X,a) und f(b,Y,c) sind nicht unifizierbar, da die Arität der beiden Terme verschieden ist, nämlich 2 und 3.

Ein Prologprogramm wird gestartet, indem eine Anfrage gestellt wird. Wie schon auf Seite 6 erwähnt, ist eine Anfrage eine Klause ohne Kopf. Der Aufruf eines Programms beginnt mit einer Anfrage der Form $\leftarrow A_1 \ldots A_n$. Nun wird, folgend einer links-nach-rechts Reihenfolge, das erste Ziel A_1 ausgewählt und versucht, diesen Term mit dem Kopf eines Prädikates des Programms zu unifizieren. Gelingt dies, so wird dieser Ausdruck durch den Rumpf der gewählten Klause ersetzt und wir erhalten:

$$\leftarrow A_{11}, \ldots, A_{1n_1}, A_2, A_3, \ldots, A_n.$$

wobei $A_{11}, \ldots, A_{1n_1}$ die Ziele des Rumpfs der Klause A_1 sind. Die Klause A_1 wurde reduziert. Eine Lösung wird gefunden, wenn alle Ziele reduziert werden können.

Eine Klause, deren Kopf mit einem aktuellen Ziel unifizierbar ist, wird gemäß ihrer Reihenfolge in der Definition des Prädikates ausgewählt. Ist keine Klause unifizierbar, so scheitert das Ziel.

Scheitert ein Ziel einer Klause, wird versucht, durch Rücksetzen in der Berechnung doch noch eine Lösung für die Anfrage zu finden. Dies wird als Backtracking bezeichnet und an folgendem Beispiel erläutert.

K_1: q(a).
K_2: q(b).
K_3: r(b).
K_4: p(X) $\leftarrow$ q(X),r(X).

Die Anfrage sei $\leftarrow$ p(X). Die Anfrage besteht also nur aus einem zu erfüllenden Ziel. Für p(X) wird eine unifizierbare Regel des Prädikates p/1 gefunden, sodaß p(X) durch den Rumpf dieser Regel ersetzt wird. Wir erhalten $\leftarrow$ q(X),r(X). Nun wird, folgend der links-nach-rechts Reihenfolge bei Auswahl von Zielen, versucht, q(X) mit dem Kopf eines Prädikates zu unifizieren. Das Prädikat q/1 besteht aus zwei Fakten. Gemäß der Reihenfolge in der Prädikatsdefinition wird das erste Faktum ausgewählt und q(X) mittels q(a) mit der Substitution θ={X/a} reduziert. Die Variable X ist nun an den Term a/1 gebunden und wir erhalten $\leftarrow$ r(a). Nun ist r(a) zu beweisen. Dies scheitert, da kein Kopf mit diesem Term unifizierbar ist. An dieser Stelle wird versucht, durch Rücksetzen in der Berechnung doch noch eine Lösung zu finden.
Es wird zurückgesetzt zur letzten Stelle, an der mehrere Möglichkeiten für die Auswahl einer Regel bestanden haben. Diese Stelle in der Berechnung wird daher als *Wahlpunkt* bezeichnet und ebenso die gleichnamige Datenstruktur, die an dieser Stelle im abstrakten Interpreter erzeugt wird (siehe 5.1.5 auf Seite 70). Eine Wahlmöglichkeit bestand bei der Auswahl des Faktums q(a). Die Auswahl wird nun rückgängig gemacht, und die Bindung der Variable X aufgehoben. Dann wird versucht, eine weitere Klause in der Definition des Prädikates zu finden, deren Kopf mit dem Ziel q(X) unifizierbar ist. Die zweite Klause des Prädikates q/1 ist unifizierbar mit der Substitution θ={X/b} und q(X) wird reduziert. Wir erhalten $\leftarrow$ r(b). Das Ziel, r(b) wird mittels der Klause K_3 reduziert und wir erhalten als Lösung für die Anfangs gestellte Anfrage: X=b.

Diese Lösung wurde also durch Backtracking gefunden. Die Reihenfolge der Antwortsubstitutionen wird durch die Reihenfolge der Auswahl der Klausen und Ziele festgelegt. Prolog führt eine Tiefensuche durch, wobei die Ziele im Rumpf einer Klause immer von links nach rechts abgearbeitet werden, was dem logischen UND entspricht. Besteht ein Prädikat aus mehreren Regeln, so entspricht dies dem logische ODER. D.h., jedem ODER entspricht ein Wahlpunkt in der Berechnung.

Vom deklarativen Standpunkt aus sollte Prolog entweder Lösungen finden oder scheitern. Wegen Prologs Auswahlverfahren treten in manchen Fällen Endlosableitungen auf. In diesen Fällen ist die deklarative Bedeutung des Programms nicht gleich der prozeduralen Bedeutung.

Folgendes Prädikat p/2 aus [Llo87] ist ein Beispiel für eine Endlosableitung. Es beschreibt eine kommutative und transitive Beziehung. Deklarativ gesehen müßte p(a,c) ableitbar sein.

K_1: p(a,b).
K_2: p(c,b).
K_3: p(X,Y) ← p(Y,X).
K_4: p(X0,X) ← p(X0,X1),p(X1,X).

Aufgrund der Endlosableitung durch Klause 3 kommt Klause 4 nie zur Anwendung. Um p(a,c) zu beweisen, müßten aber beide Regeln zur Anwendung kommen. Auch wenn die Reihenfolge der Regeln geändert wird, ist es nicht möglich, p(a,c) zu beweisen. Vertauschen wir etwa die Klausen 3 und 4, sodaß die transitive Beziehung vor der kommutativen Beziehung definiert wird. Die Anwendung der transitiven Beziehung führt ebenfalls zu einer Endlosableitung, sodaß die kommutative Beziehung nicht für den Beweis verwendet wird. D.h., wie auch immer die Klausen angeordnet werden: p(a,c) kann nicht abgeleitet werden. Endlosableitungen verändern also die deklarative Bedeutung eines Programms.

Im Folgenden werden einige häufig verwendete Datenstrukturen vorgestellt.

2.3.2 Akkumulatoren

In Prolog muß für jedes Zwischenergebnis einer Berechnung eine Variable reserviert werden. Es ist nicht möglich, Speichervariablen während einer Berechnung zu verändern. Daher werden Prädikate um Argumente erweitert, die diese Zwischenergebnisse aufnehmen können. Diese Argumente heißen Akkumulatoren.

Als Beispiel wird das Summieren von Elementen einer Liste von Zahlen angeführt [SS86]. Das Prologprädikat ohne Akkumulatoren sieht folgendermaßen aus:

```
liste_summe([],0).
liste_summe([X|Xs],Summe) ←
        liste_summe(Xs,Summe0),
        Summe is X+Summe0.
```

Die leere Liste hat Summe 0. Die Summe des Rests Xs der Liste wird zum Wert des Kopfs summiert. Das Aufsummieren der Elemente wird beim letzten Element der Liste begonnen. Es ist daher nicht möglich während der Berechnung auf Teilsummen zuzugreifen, die beim ersten Element beginnen.

Wird dieses Prädikat um einen Akkumulator erweitert, wird die Summe beginnend beim ersten Element berechnet. Alle Teilsummen stehen während der Berechnung zur Verfügung.

```
liste_summe(Xs,Summe) ←
        liste_akkumulator_summe(Xs,0,Summe).

liste_akkumulator_summe([],Summe,Summe).
liste_akkumulator_summe([X|Xs],T0,Summe) ←
        T is T0+X,
        liste_akkumulator_summe(Xs,T,Summe).
```

Das Prädikat liste_summe/2 ist das Interface, das die neue Version in das Programm einbindet. Das zweite Argument, der Akkumulator, wird mit dem Wert Null initialisiert. In jeder Rekursionsebene wird der Wert eines Elements zur Summe hinzugezählt. Bei Terminierung der Berechnung wird das dritte Argument mit dem Wert der Summe unifiziert, das deswegen als Ausgabeargument bezeichnet wird.

Da durch Verwendung eines Akkumulators alle Teilsummen während der Berechnung zur Verfügung stehen, ist es möglich, alle Teilsummen in einer Liste zusammenzufassen. Dies leistet folgendes Prädikat liste_summe_teilsummen/3.

```
liste_summe_teilsummen(Xs,Summe,Teilsummen) ←
        liste_akkumulator_summe_teilsummen(Xs,0,Summe,Teilsummen).

liste_akkumulator_summe_teilsummen([],Summe,Summe,[]).
liste_akkumulator_summe_teilsummen([X|Xs],T0,Summe,[T|Teilsummen]) ←
        T is T0+X,
        liste_akkumulator_summe_teilsummen(Xs,T,Summe,Teilsummen).
```

Die Teilsummen stehen in der ersten Version dieses Prädikates, liste_summe/2, während der Berechnung nicht zur Verfügung. Durch die Verwendung von Akkumulatoren kann die Effizienz von Programmen erhöht werden und auf Zwischenergebnisse in Berechnungen direkt zugegriffen werden. Sie sind daher ein häufig verwendetes Programmkonstrukt in Prologprogrammen.

Betrachten wir aber nochmals die zweite Version dieses Prädikates, liste_akkumulator_summe/3, was uns dann direkt zu einer weiteren grundlegenden Datenstruktur Prologs führt. Die Summe der einzelnen Elemente E_i für eine Liste der Länge n kann man anschreiben als $Summe{=}E_1{+}E_2{+}\ldots{+}E_n$. Jede Teilsumme läßt sich aber auch mit Hilfe der Summe beschreiben. So ist etwa das erste Element gleich der Differenz $Summe{-}E_2{+}\ldots{+}E_n$. Dieses Prinzip, mit Differenzen zu arbeiten, findet man in Prolog bei der Verwendung von Listendifferenzen. In Analogie zu den hier erwähnten Teilsummen wird mit Teillisten operiert.

2.3.3 Listendifferenzen

Werden Listendifferenzen anstatt normaler Listen verwendet, kann die Effizienz von Prologprogrammen erhöht werden. Der Vorteil von Listendifferenzen ist der, daß das Aneinanderhängen von zwei Listendifferenzen in konstanter Zeit durchgeführt werden kann, wohingegen normale Listen nur in linearer Zeit - proportional zur Länge der Liste - aneinandergehängt werden können.

Definition. Gegeben seien zwei Listen A und B. Dann heißt die Differenz dieser Listen, geschrieben als A-B, $Listendifferenz$.

Jede Liste ist durch verschiedene Listendifferenzen darstellbar. Die Liste [1] kann etwa durch die Differenz folgender Listen beschrieben werden:

- A=[1,2,3],B=[2,3]
- A=[1], B=[]
- A=[1,|Xs], B=Xs

Zwei Listendifferenzen werden durch folgendes Faktum in konstanter Zeit aneinandergehängt:

```
ld_ld_aneinander(Xs0−Xs1,Xs1−Xs,Xs0−Xs).
```

Die Listendifferenz Xs0−Xs ist das Ergebnis des Aneinanderhängens von Xs0−Xs1 und Xs1−Xs. Die Variable Xs1 tritt in beiden Listendifferenzen auf. Zwei Listendifferenzen As0−As und Bs0−Bs sind daher nur dann aneinanderzuhängen, wenn As und Bs0 unifizierbar sind. Da eine Liste durch verschiedene Listendifferenzen dargestellt werden kann, ist es entscheidend, die „richtige“ Darstellung zu wählen.

Gegeben seien etwa die Listen [a] und [b,c]. Die Darstellungen [a]-[] und [b,c]-[] sind mittels dl_dl_aneinander/3 nicht aneinanderzuhängen, obwohl sie die Listen [a] und [b,c] als Differenzen darstellen. Wählen wir aber [a|Xs]-Xs und [b,c]-[], so erhalten wir die Listendifferenz [a,b,c]−[], die der Liste [a,b,c] entspricht.

Zusammenfassung

In diesem Kapitel haben wir gesehen, daß Prolog sowohl eine deklarative als auch eine prozedurale Semantik besitzt. Die prozedurale Semantik kam in der Auswahlreihenfolge von Klausen und Zielen zum Ausdruck. Aufgrund dieser festgelegten Reihenfolge konnte es zu Endlosableitungen kommen, sodaß die deklarative Bedeutung eines Programms verändert wurde. Wir sahen, daß die Unifikation und Backtracking zwei wesentliche Elemente Prologs sind. Die Unifikation entsprach der

Zuweisung und der Argumentübergabe in imperativen Programmiersprachen. Backtracking wurde angewendet, um bei Scheitern eines Ziels durch Rücksetzen in der Berechnung doch noch eine Lösung zu finden. Wir sahen, daß Variablenbindungen gelöst und neue Klausen gewählt werden, um eine Anfrage mittels Backtracking zu beweisen.

Wir erläuterten zwei Programmkonstrukte, die in Prologprogrammen sehr häufig Verwendung finden: Akkumulatoren und Listendifferenzen. Akkumulatoren wurden verwendet, um iterative Algorithmen effizient umzusetzen und um auf Zwischenergebnisse bei Berechnungen direkt zugreifen zu können. Die Verwendung von Listendifferenzen erwies sich als eine interessante Alternative, da Listendifferenzen in konstanter Zeit aneinandergehängt werden konnten.

Daher wurden Transformationen entwickelt, die Listen durch Listendifferenzen ersetzen. Dies wird im folgenden Kapitel in 3.2.3 erläutert. Programme, in denen Listendifferenzen auftreten, werden durch die in Kapitel 4 umfassend dargestellten Kontinuationstransformationen optimiert.

Kapitel 3

Programmtransformationen

Wir beschäftigen uns in diesem Kapitel mit der Transformation von logischen Programmen. Das Ziel ist, die Effizienz von Programmen zu erhöhen. Diese Techniken nehmen in der Software-Entwicklung einen hohen Stellenwert ein.

In vielen Fällen ist ein optimiertes Programm schlecht lesbar. Der Gewinn an Effizienz, der durch Optimierungen erreicht werden kann, steht oft guter Lesbarkeit diametral gegenüber. Die Vereinigung dieser Gegensätze wird durch Programmtransformationen erreicht. Das Programm kann ausschließlich in Hinsicht auf Übersichtlichkeit und Klarheit formuliert werden. Dann wird das Programm in ein effizienteres transformiert und zeigt das gewünschte Laufzeitverhalten.

Die hier erläuterten Transformationssysteme bestehen aus Regeln, gemäß denen das Programm transformiert wird, und Strategien, welche eine entsprechende Folge von Anwendungen der Transformationsregeln vorschreiben. Jede Anwendung einer Regel entspricht einem Schritt in der Transformation des Programms. Da nicht jede Folge von Regelanwendungen zu einem optimierten Programm führt, sind Strategien notwendig. Dieser „Regeln+Strategien"-Ansatz wurde erstmals von Burstall und Darlington [BD77] vorgestellt.

Definition. Ein Programmtransformationsprozeß, beginnend mit einem Programm P_0, ist eine Folge von Programmen $P_0 \dots P_n$ (genannt $Transformationsfolge$). Im Zuge dieses Transformationsprozesses gelangt man von Programm P_k zu P_{k+1}, mit $0 \leq k \leq n-1$, indem eine Transformationsregel angewendet wird, die von den Programmen $P_0 \dots P_k$ abhängen kann.

Im Folgenden wird die Menge der Variablen, die in einem Term T auftreten, mit $vars(T)$ bezeichnet.

3.1 Transformationsregeln

Die zwei grundlengenden Transformationsregeln sind das Entfalten und Falten von Klausen. Dieser Ansatz wurde auf logische Programme erstmals von Tamaki&Sato engewendet [TS84]. Falten wird als die inverse Transformation zu Falten angesehen. Dies trifft auch auf die hier verwendeten Definitionen zu. Wir werden sehen, daß uns die Falten-Regel erlaubt, die rekursive Struktur eines Programms zu verändern. Dadurch wird es möglich sein, die Effizienz von Programmen entscheidend zu verbessern.

3.1.1 Entfalten

Definition. Sei P_k ein Programm $\langle E_1, \ldots, E_r, K, E_{r+1}, \ldots, E_s\rangle$ und sei K eine Klause H← α, A, β. A ist ein Literal. α und β sind (möglicherweise leere) Folgen von Literalen. Weiters seien folgende Bedingungen erfüllt:

1. $\langle D_1, \ldots, D_n\rangle$, mit $n \geq 1$, ist eine Teilfolge aller Klausen in einem Programm P_j, mit $0 \leq j \leq k$, sodaß A unter Anwendung des jeweiligen allgemeinsten Unifikators $\theta_1, \ldots, \theta_n$ mit Kopf(D_1),...,Kopf(D_n) unifizierbar ist.
2. K_i ist die Klause (H← α,Rumpf(D_i), β)θ, für $i = 1 \ldots n$.

Wenn wir K hinsichtlich A unter Verwendung von $D_1, \ldots, D_n$ in P_j entfalten, dann erhalten wir die Klausen $K_1, \ldots, K_n$ und somit ein neues Programm $P_{k+1} = \langle E_1, \ldots, E_r, K_1, \ldots, K_n, E_{r+1}, \ldots, E_s\rangle$.

Das folgende Beispiel zeigt eine sehr einfache Anwendung des Entfaltens, wie sie in ähnlicher Form auch in imperativen Programmiersprachen zur Verminderung der Anzahl der Prozeduraufrufe verwendbar ist. Gegeben seien folgende zwei Klausen

P_0:

K_1: p(X0,X) ← q(X0,X1),s(X1,X).
K_2: q(X0,X) ← r(X0,X1),r(X1,X).

Durch Entfalten von Klause K_1 hinsichtlich q/2 unter Verwendung von K_2 erhalten wir

P_1:

K_3: p(X0,X) ← r(X0,X1),r(X1,X2),s(X2,X).

Das neue Programm P_1 besteht nur mehr aus der Klause K_3. Diese enthält zum Unterschied von K_1 nicht mehr den Aufruf von q/2, sondern stattdessen den Rumpf der Klause K_2.

Im folgenden Beispiel sind verschiedene Substitutionen nötig.

P_0:

K_1: p(X,Y) $\leftarrow$ q(f(X),X1), r(X1,X).
K_2: q(f(a),c) $\leftarrow$ q_1(e), q_2(e).
K_3: q(f(b),d) $\leftarrow$ q_1(X2), q_2(X2).

Durch Entfalten von Klause K_1 hinsichtlich $q/2$ unter Verwendung von K_2 und K_3 erhalten wir

P_1:

K_4: p(a,c) $\leftarrow$ q_1(e), q_2(e), r(c,a).
K_5: p(b,d) $\leftarrow$ q_1(X2), q_2(X2), r(d,b).

Hier ist $\theta_1 \neq \theta_2$. In Programm P_0 bestand das Prädikat q/2 aus zwei Klausen. Durch Anwendung der Falten-Regel wurde der Aufruf von q/2 beseitigt und die Anzahl der Klausen von drei auf zwei verringert.

3.1.2 Entfaltungsbaum

Im allgemeinen gibt es mehrere Möglichkeiten die Falten-Regel anzuwenden. Es ist daher naheliegend dies in Form eines Baumes darzustellen. Die Knoten enthalten die Klausen und die Zweige sind durch die Ziele festgelegt sind, hinsichtlich derer entfalten wird.

Definition. Gegeben sei ein Programm P und eine Klause K. Dann steht in der Wurzel die Klause C. Durch Entfalten-Schritte werden Nachfolger erzeugt. Ein Knoten enthält entweder keine Nachfolger oder er hat $n \geq 1$ Nachfolger, die durch Entfalten der Klause dieses Knotens hinsichtlich ihrer Ziele erzeugt werden.

Die Konstruktion eines Entfaltungsbaumes ist nicht-deterministisch. Während der Generierung des Baumes muß immer wieder entschieden werden, ob ein Knoten Nachfolger haben soll oder nicht, d.h., ob eine Entfaltungsregel angewendet wird. Wenn das der Fall ist, muß entschieden werden, hinsichtlich welchen Ziels entfaltet wird.

Gegeben sei folgendes Programm P, das dem Programm in 3.1.1 entspricht, dem aber noch die Klausen K_3 bis K_6 hinzugefügt wurden.

P_0:

K_1: p(X0,X) $\leftarrow$ q(X0,X1),s(X1,X).
K_2: q(X0,X) $\leftarrow$ r(X0,X1),r(X1,X).
K_3: q(c,c).
K_4: r(a,b).
K_5: r(b,a).
K_6: s(X,X).

Der Entfaltungsbaum für die Klause p(X0,X) $\leftarrow$q(X0,X1),s(X1,X) ist in Abbildung 3.1 dargestellt. Dabei wurde eine links-nach-rechts Klausenauswahl verwendet und jene Ziele, hinsichtlich derer entfaltet wurde, sind unterstrichen dargestellt.

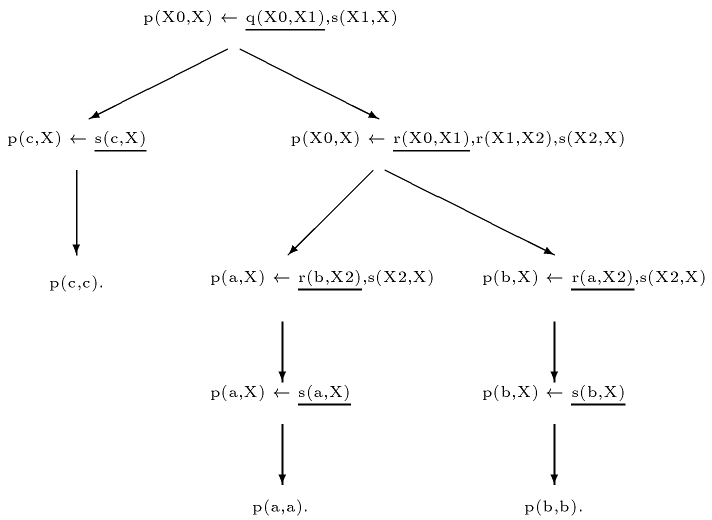

Abbildung 3.1: Der Entfaltungsbaum für $\langle$ P_0, p(X0,X) $\leftarrow$ q(X0,X1), s(X1,X) $\rangle$

3.1.3 Falten

Durch Anwendung von Falten und Entfalten auf ein Programm können redundante Berechnungen beseitigt werden. Falten ist die inverse Regel zum Entfalten und umgekehrt, wenn für die Transformation die Klausen jedes Programms der Transformationsfolge verwendet werden können [PP94]. Es muß daher in einem Transformationsprozeß durch eine Strategie sichergestellt werden, daß immer neue Programme erzeugt werden.

Definition. Sei P_k ein Programm $P_k = \langle E_1, \ldots, E_r, K_1, \ldots, K_n, E_{r+1}, \ldots, E_s \rangle$ und sei $\langle D_1, \ldots, D_n \rangle$ eine Folge von Klausen in einem Programm P_k, mit $0 \leq j \leq k$. Nehmen wir weiters an, daß ein Literal A existiert, sodaß für i=1... n gilt:

1. Kopf(D_i) ist unifizierbar mit A mittels eines allgemeinsten Unifikators θ_i.
2. K_i ist eine Klause ($H \leftarrow \alpha$,Rumpf(D_i),β)θ_i, wobei α und β Folgen von Literalen sind.
3. Für keine Klause K von P_i, die in der Folge $\langle D_1 \dots D_n \rangle$ nicht vorkommt, ist Kopf(K) mit A unifizierbar.

Wenn wir $K_1 \dots K_n$ unter Verwendung von $D_1 \dots D_n$ in P_j falten, dann erhalten wir die Klause H $\leftarrow$ α, A, β, genannt K, und erhalten ein neues Programm $P_{k+1} = \langle E_1, \dots, E_r, K, E_{r+1}, \dots, E_s \rangle$.

Folgendes Beispiel zeigt das Falten zweier Klausen K_1 und K_2 unter Verwendung von zwei Klausen D_1 und D_2.

P_0:

K_1: p(f(f(X),Y)) $\leftarrow$ r(X),s(X),t(Y).
K_2: p(g(g(X),g(Y))) $\leftarrow$ u(X),v(X),t(Y).
D_1: a(f(f(X),Y),Y) $\leftarrow$ r(X),s(X).
D_2: a(g(g(X),g(Y)),Y) $\leftarrow$ u(X),v(X).

Literal A sei a(Z,Y).

Für das Literal a(Z,Y) sind alle drei Bedingungen erfüllt:

1. Die Köpfe der beiden Klausen D_1 und D_2 sind mit a(Z,Y) unifizierbar. Die Substitutionen sind θ_1={Z/f(f(X),Y)} und θ_2={Z/g(g(X),g(Y))}.
2. Rumpf(D1) tritt im Rumpf von K_1 auf und Rumpf(D2) im Rumpf von K_2. α ist leer und β=t(Y).
3. Die beiden Klausen K_1 und K_2 sind nicht mit a(Z,Y) unifizierbar. D_1 und D_2 sind die zu faltenden Klausen. Sonst existieren keine Klausen in P_0. Daher ist auch diese Bedingung erfüllt.

Das Falten kann man in zwei Schritten darstellen (formal ist es ein Schritt). Zuerst werden die Ziele r/1 und s/1 bzw. u/1 und r/1 durch den jeweiligen Kopf von a/2 ersetzt:

p($\underbrace{f(f(X),Y)}_{Z}$) $\leftarrow$ a($\underbrace{f(f(X),Y)}_{Z}$,Y),t(Y).
also $\theta_1 = \{Z/f(f(X),Y)\}$

p($\underbrace{g(g(X),g(Y))}_{Z}$) $\leftarrow$ a($\underbrace{g(g(X),g(Y))}_{Z}$,Y),t(Y).
also $\theta_2 = \{Z/g(g(X),g(Y))\}$

Dann werden die Substitutionen θ_1 bzw. θ_2 durchgeführt. Wir erhalten somit nach Falten von K_1 und K_2 hinsichtlich r(X),s(X) bzw. u(X),v(X) unter Verwendung von D_1 bzw. D_2 mit θ_1 bzw. θ_2 das Programm

P_1:
K_3: p(Z) $\leftarrow$ a(Z,Y),s(Y).
D_1: a(f(f(X),Y),Y) $\leftarrow$ r(X),s(X).
D_2: a(g(g(X),g(Y)),Y) $\leftarrow$ u(X),v(X).

Die Prädikate p/2 und a/2 bestanden im Programm P_0 aus jeweils zwei Klausen. Die Anzahl der Wahlmöglichkeiten für eine Klause in der Definition eines Prädikates ist ein entscheidender Faktor für die Effizienz eines Programms. Jeder Wahlmöglichkeit entspricht ein Wahlpunkt in Prolog. Die Anzahl der Wahlpunkte konnten durch Falten um eins verringert werden und die Effizienz des Programms somit erhöht werden.

3.1.4 Selbstfalten

Wird zum Falten einer Klause die Klause selbst verwendet, so kann sich das Terminationsverhalten der Klause (bzw. des Prädikates) ändern. Falten wir z.B. die Klause $p \leftarrow q$ mit sich selbst, so erhalten wir $p \leftarrow p$. Die abgeleitete Klause ist nun rekursiv und terminiert nicht - zum Unterschied von der ursprünglichen Klause.

In Transformationssystemen findet diese Art des Faltens keine Anwendung, da dadurch die Bedeutung von Programmen verändert wird.

3.1.5 T&S-Falten

Die von Tamaki&Sato vorgestellte Falten-Regel [TS84] ist eine Instanz des in 3.1.3 beschriebenen Faltens. Der wesentliche Unterschied besteht darin, daß mittels T&S-Falten mit genau einer Klause gefaltet wird, die nicht rekursiv sein darf.

Definition. Sei P_k ein Programm $P_k = \langle E_1, \ldots, E_r, K, E_{r+1}, \ldots, E_s\rangle$ und sei D

eine Klause im Programm P_j, mit $0 \leq j \leq k$. Weiters gelten folgende Bedingungen:

1. K ist eine Klause H$\leftarrow$ α,Rumpf(D)θ,β derart, daß α, Rumpf(D)θ und β Folgen von Literalen sind,

2. Die Substitution θ, die auf Menge vars(Rumpf(D))-vars(Kopf(D)) beschränkt ist, ist eine Variablenumbenennung, deren Bild eine leere Schnittmenge mit der Menge vars(H,α,Kopf(D)θ,β) hat, und

3. das Prädikatensymbol von Kopf(D) tritt in P_k nur einmal auf (d.h. D ist nicht rekursiv).

Wenn wir K hinsichtlich Rumpf(D)θ unter Verwendung von D in P_k T&S-Falten, erhalten wir eine Klause $H\leftarrow$ α,Kopf(D)θ,β, genannt $K_{T\&S}$. Wir erhalten ein neues Programm P_{k+1} indem wir K durch $K_{T\&S}$ in P_k ersetzen.
Bei Anwendung der T&S-Falten Regel unterscheidet sich das abgeleitete Programm P_{k+1} von P_k nur in genau einer Klause.

Das folgende Beispiel zeigt, daß durch Anwendung von Entfalten und darauffolgendem T&S-Falten ein Prädikat, das ursprünglich scheiterte, nach erfolgter Transformation nicht mehr terminiert, d.h. das Terminationsverhalten kann durch Anwendung von T&S-Falten verändert werden. Gegeben sei also das Programm

P_0:

K_1: p(X) $\leftarrow$q(X),r(X).
K_2: q(a).
K_3: r(b) $\leftarrow$r(b).

Nach Entfalten von K_1 hinsichtlich r(X) unter Verwendung von K_3 erhalten wir

P_1:

K_4: p(b) $\leftarrow$q(b),r(b).
K_2: q(a).
K_3: r(b) $\leftarrow$r(b).

Aber nach T&S-Falten von K_4 hinsichtlich Rumpf(K_4) unter Verwendung von K_1 erhalten wir

P_2:

K_5: p(b) $\leftarrow$p(b).
K_2: q(a).
K_3: r(b) $\leftarrow$r(b).

P_2 terminiert für die Anfrage $\leftarrow p(b)$ nicht und hat somit ein unterschiedliches Terminationsverhalten gegenüber P_0, das für diese Anfrage scheiterte.

3.1.6 Definition

Einführen einer Definition

Um eine effizientere Definition eines Prädikates herleiten zu können, wird eine neues Prädikat eingeführt. Durch Anwendung von Regeln wie etwa Falten und Entfalten wird dann eine effizienteres Programm erhalten. Um die Definition dieses neuen Prädikates angeben zu können, bedarf es bestimmter Heuristiken. Dieses Prädikat wird in der Literatur als 'Eureka-Prädikat' bezeichnet.

Definition. Wir können ein Programm P_{k+1} erhalten, indem P_k mit einer Folge von Klausen p(...)$\leftarrow$ Rumpf$_i$ | i=1 ... n erweitert wird, wobei das Prädikatensymbol p in $P_0 \ldots P_k$ nicht auftritt [PP94] (d.h., es wird ein neues Prädikat hinzugefügt).

T&S-Definition

Der Unterschied der T&S-Definition zur oben erwähnten Definition besteht darin, daß nur genau eine nicht-rekursive Klause neu definiert werden kann. Diese Definition wurde von Tamaki und Sato in [TS84] und [TS86] vorgeschlagen.

Im Folgenden seien alle Prädikate als *neu* bezeichnet, wenn sie genau einmal im Kopf einer Klause des Startprogramms P_0 auftreten oder mittels der T&S-Definition Regel eingeführt wurden; alle anderen Prädikate werden als *alt* bezeichnet.

Definition. Gegeben sei eine Transformationsfolge $P_0 \ldots P_k$. Wir können ein neues Programm P_{k+1} erhalten, indem zu Programm P_k eine Klause H $\leftarrow$ Rumpf derart hinzugefügt wird, daß folgende Bedingungen erfüllt sind:

1. Das Prädikat von H tritt in $P_0 \ldots P_k$ nicht auf und
2. Rumpf besteht aus Literalen mit alten Prädikaten, die in $P_0 \ldots P_k$ auftreten.

Die neu hinzugefügte Klause H $\leftarrow$ Rumpf heißt $T\&S\text{-}Definition$.

Beseitigen einer Definition

Definition. Wir können ein Programm P_{k+1} erhalten, indem in Programm P_k alle Klausen der Definition eines Prädikates p beseitigt werden. Das Prädikat p darf in P_0 nicht auftreten. Weiters darf kein anderes Prädikat von p in P_k abhängig sein (d.h. ein Prädikat, das nicht in P_0 enthalten war, wird beseitigt).

Das Beseitigen einer Definition kann als das Inverse des Einführens einer Definition angesehen werden [PP94].

3.2 Transformationsstrategien

Eine Transformationsstrategie besteht aus Metaregeln, die es ermöglichen, jene Folge von Regeln festzulegen, durch deren Anwendung man, ausgehend von Programm P_0, ein effizienteres Programm P_n erhält. Im Folgenden werden drei grundlegende Strategien vorgestellt. Durch Anwendung dieser ist es etwa möglich Variablen zu beseitigen, Listendifferenzen einzuführen oder auch Rekursion in Iteration überzuführen. Wir beginnen mit der Definition der drei Strategien Tupeln, Schleifenabsorption und Generalisation. Diese werden dann an Hand von Beispielen ausführlich erläutert. Abschließend wird darauf eingegangen, in welchen Fällen die Transformationen die Semantik von Prologprogrammen erhalten.

Wir geben hier eine Definition von unnötigen Variablen an, wie sie auch in [PP91] verwendet wird. Dieser Begriff wird im Folgenden benötigt, um die Anwendbarkeit von Transformationsregeln zu definieren.

Definition. Gegeben sei eine Klause K der Form $H\leftarrow A_1 \ldots A_m, B_1 \ldots B_n$. Dann heißen die Variablen der Folge $A_1 \ldots A_m$ von Atomen, die in vars($A_1 \ldots A_m$) $\cap$ vars(H,$B_1 \ldots B_n$) enthalten sind, *verbindende* Variablen [PP91].

Für Variablen, die in mehreren Zielen auftreten, verwendet wir auch den Begriff *gemeinsame* Variable.

3.2.1 Grundlegende Transformationsstrategien

Tupeln

Definition. Seien $A_1 \ldots A_n$, mit $n \geq 1$, Atome im Rumpf einer Klause K eines gegebenen Startprogramms. Wir führen ein neues Prädikat p_neu ein, definiert durch eine Klause T der Form

$$\text{p\_neu}(X_1 \ldots X_k)\leftarrow A_1 \ldots A_n$$

wobei $X_1 \ldots X_k$ verbindende Variablen von $A_1 \ldots A_n$ in K sind. Wir versuchen dann die rekursive Definition des Eureka-Prädikates p_neu/k zu finden, indem wir zuerst mehrmals die Entfalten-Regel anwenden und anschließend die passende Falten-Regel unter Verwendung der Klause T. Schlußendlich falten wir die Klause K hinsichtlich der Atome $A_1 \ldots A_n$ unter Verwendung der Klause T.

Die Tupeln-Strategie kann angewendet werden, wenn in einer Klause verbindende

Variablen auftreten. Es werden redundante Berechnungen, die durch die Ziele in dieser Klause mehrmals ausgewertet werden, beseitigt. Einige der Zwischenvariablen können ebenfalls beseitigt werden.

Schleifenabsorption

Definition. Gegeben sei die Klause K eines Entfaltungsbaumes der Form $H \leftarrow A_1 \ldots A_m, B_1 \ldots B_n$. Diese Klause stehe nicht in der Wurzel des Baums. Weiters sei der Rumpf eines Nachkommens D der Klause K gegeben. Rumpf(D) enthält (als Teilfolge von Atomen) eine Instanz $(A_1 \ldots A_m)\theta$ von $A_1 \ldots A_m$ für einige Substitutionen θ. Nehmen wir weiters an, daß die Klausen im Pfad des Entfaltungsbaums von K nach D nicht durch Anwendung von Transformationsregeln auf $B_1 \ldots B_n$ generiert worden sind. Wir führen ein neues Prädikat ein, das durch folgende Klause A definiert ist:

$$\text{p\_neu}(X_1 \ldots X_k) \leftarrow A_1 \ldots A_m$$

Die Menge $\{X_1 \ldots X_k\}$ ist die kleinste Untermenge von vars$(A_1 \ldots A_m)$, die notwendig ist, um die beiden Klausen K und D unter Verwendung einer Klause, deren Rumpf $A_1 \ldots A_m$ ist, zu falten. Wir falten Klause K unter Verwendung von D und suchen die rekursive Definition des Eureka-Prädikates p_neu/k. Diese kann gefunden werden, indem Entfalten-Schritte durchgeführt werden, die den Schritten von Klause K zu D im Entfaltungsbaum entsprechen. Abschließend falten wir nochmals unter Verwendung von Klause A [PP94].

Die Schleifenabsorption wird auch beim Tupeln angewendet.

Generalisation

Definition. Gegeben sei eine Klause K der Form $H \leftarrow A_1 \ldots A_m, B_1 \ldots B_n$. Wir definieren ein neues Prädikat gen_p mittels einer Klause G der Form

$$\text{gen\_p}(X_1 \ldots X_k) \leftarrow GenA_1 \ldots GenA_m$$

wobei $(GenA_1 \ldots GenA_m)\theta = A_1 \ldots A_m$, für eine gegebene Substitution θ. Die Menge $\{X_1 \ldots X_k\}$ ist eine Obermenge der Variablen, die notwendig sind, um K unter Verwendung einer Klause zu falten, deren Rumpf $GenA_1 \ldots GenA_m$ ist. Dann falten wir K unter Verwendung von G und erhalten

$$H \leftarrow \text{gen\_p}(X_1 \ldots X_k)\theta,\ B_1 \ldots B_n$$

Schlußendlich muß noch die rekursive Definition des Eureka-Prädikates gen_p gefunden werden [PP94].

Während der Programmtransformation wird ein Entfaltungsbaum erzeugt. Eine passende Form für die neu eingeführte Klause kann man erhalten, indem Klause K mit einem ihrer Nachkommen im Entfaltungsbaum unifiziert wird.

3.2.2 Aufbauende Techniken

Beseitigung unnötiger Variablen

In [PP91] wird eine Strategie vorgestellt, mit der unnötige Variablen beseitigt werden können. Es werden die Regeln Entfalten, Definition und Falten (in dieser Reihenfolge) iterativ angewendet, um ein effizienteres Programm zu erhalten. Wir geben zuerst die Definition einer unnötigen Variable an und erläutern dann die Beseitigung unnötiger Variablen an Hand eines Beispiels. Die Vollständigkeit der Transformationen wurde in [PP94b] bewiesen.

Definition. Eine Variable X einer Klause K heißt *unnötig*, wenn zumindest eine der beiden folgenden Bedingungen erfüllt ist:

- X tritt mehr als einmal im Rumpf von K auf (auch gemeinsame Variable genannt).
- X tritt nicht im Kopf von K auf (auch existentielle Variable genannt).

Um die Semantik von logischen Programmen als eine Relation zwischen Regeln und Anfragen zu beschreiben, sind mehrfach auftretende Variablen im Rumpf einer Klause und existentielle Variablen nicht nötig. In diesem Sinne werden gemeinsame, existentielle und verbindende Variablen *unnötig* genannt. Sie treten auch in Programmen auf, die Datenstrukturen mehrfach durchsuchen.

Folgendes Beispiel zeigt die Beseitigung von unnötigen Variablen mittels der Regeln Falten, Definition und Entfalten. Die Strategien Tupeln und Generalisation werden verwendet.

Gegeben sei das Programm P_0. Die Variable EinBaum ist ein binärer Baum, in dessen Blättern Werte eingetragen sind. Es wird der kleinste Wert Min dieser Blätter ermittelt und ein neuer Baum AusBaum generiert, in dessen Blättern alle Werte durch diesen kleinsten Wert ersetzt sind. Dieses Programm sieht folgendermaßen aus:

P_0:

K_1: baum_minbaum(EinBaum,AusBaum) ←
    baum_min(EinBaum,Min),
    ersetzt(Min,EinBaum,AusBaum).
K_2: baum_min(blatt(X),X).
K_3: baum_min(baum(L,R),Min) ←
    baum_min(L,MinL),
    baum_min(R,MinR),
    wert_wert_min(MinL,MinR,Min).
K_4: ersetzt(Min,blatt(_),blatt(Min)).
K_5: ersetzt(Min,baum(EinL,EinR),baum(AusL,AusR) ←
    ersetzt(Min,EinL,AusL),
    ersetzt(Min,EinR,AusR).

wobei $wert\_wert\_min(W1, W2, M)$ gilt, wenn M das Minimum von $W1$ und $W2$ ist.

Das Programm P_0 durchwandert den gesamten Baum zweimal, da die beiden Prädikate baum_min/2 und ersetzt/3 den Baum je einmal durchsuchen. Die Variable InBaum in Klause K_1 ist eine verbindende Variable. Diese ist unnötig und kann mit der Tupeln-Strategie entfernt werden. Es ist hier nicht notwendig, eine neue Definition einzuführen, da beide Prädikate, die getupelt werden sollen, den gesamten Rumpf der Klause K_1 definieren. Daher muß nur die rekursive Definition des Prädikates baum_minbaum gefunden werden. Zu Beginn müssen einige Entfalten-Schritte durchgeführt werden.

Es wird Klause K_1 hinsichtlich baum_min/2 unter Verwendung der Klause K_2 bzw. K_3 entfaltet, wodurch wir folgende Klausen erhalten:

K_6: baum_minbaum(blatt(Min),BaumAus) ←
    ersetzt(Min,blatt(Min),BaumAus).
K_7: baum_minbaum(baum(L,R),BaumAus) ←
    baum_min(L,MinL),
    baum_min(R,MinR),
    wert_wert_min(MinL,MinR,Min),
    ersetzt(Min,baum(L,R),BaumAus).

Diese beiden Klausen werden nochmals entfaltet; Klause K_6 wird hinsichtlich ihres einzigen Ziels unter Verwendung von Klause K_4 entfaltet. Die Klause K_7 wird hinsichtlich ersetzt/3 unter Verwendung der Klause K_5 entfaltet und wir erhalten

K_8: baum_minbaum(blatt(Min),blatt(Min)).
K_9: baum_minbaum(baum(EinL,EinR),baum(AusL,AusR)) $\leftarrow$
    baum_min(EinL,MinL),
    baum_min(EinR,MinR),
    min(MinR,MinL,Min),
    ersetzt(Min,EinL,AusL),
    ersetzt(Min,EinR,AusR).

In Klause K_9 wurden der besseren Lesbarkeit wegen die Variablennamen L durch EinL und R durch EinR ersetzt. Entsprechend der Tupeln-Strategie, sollten nun die Ziele baum_min(EinL,MinL) und ersetzt(Min,EinL,AusL) (in Klause K_9) unter Verwendung von Klause K_1 gefaltet werden, um eine rekursive Definition von baum_minbaum zu erhalten. Dies ist allerdings nicht möglich, da die Ziele keine Instanz von baum_minbaum(InBaum,Min) und ersetzt(Min,EinBaum,AusBaum) (in Klause K_1) sind. Daher verwenden wir die Strategie Generalisation und führen eine neue Definition ein:

K_{10}: baum_minbaum2(EinBaum,MinL,MinR,AusBaum) $\leftarrow$
    baum_min(EinBaum,MinL),
    ersetzt(MinR,EinBaum,AusBaum).

Der Rumpf der Klause K_{10} ist die spezifischste Generalisation der beiden oben genannten Zielkonjunktionen, die gefaltet werden. Wird nun Klause K_1 unter Verwendung von Klause K_{10} gefaltet, so erhalten wir

K_{11}: baum_minbaum(EinBaum,AusBaum) $\leftarrow$
    baum_minbaum2(EinBaum,Min,Min,AusBaum).

Nun muß noch die rekursive Definition des Prädikates baum_minbaum2/4 gefunden werden, die mit Klause K_{10} eingeführt worden war. Zu diesem Zweck sind wieder jene Entfalten-Schritte durchzuführen, unter Anwendung derer wir die Klausen K_8 und K_9 erhalten hatten.

Es wird also Klause K_{10} hinsichtlich baum_min/2 unter Verwendung der Klause K_2 bzw. K_3 entfaltet. Wir erhalten die Klausen

K_{12}: baum_minbaum2(blatt(MinL),MinL,MinR,AusBaum) ←
ersetzt(MinR,AusBaum).
K_{13}: baum_minbaum2(baum(L,R),MinL,MinR,AusBaum) ←
baum_min(L,MinL),
baum_min(R,MinR),
wert_wert_min(MinL,MinR,Min),
ersetzt(Min,baum(L,R),AusBaum).

Nun werden diese beiden Klausen nochmals entfaltet. Klause K_{12} wird hinsichtlich ihres einzigen Ziels unter Verwendung von Klause K_4 entfaltet. Weiters wird die Klause K_7 hinsichtlich ersetzt/3 unter Verwendung von Klause K_5 entfaltet und wir erhalten folgende beiden Klausen

K_{14}: baum_minbaum2(blatt(MinL),MinL,MinR,blatt(MinR)).
K_{15}: baum_minbaum2(baum(EinL,EinR),MinL,MinR,baum(AusL,AusR)) ←
baum_min(EinL,MinL),
baum_min(EinR,MinR),
wert_wert_min(MinL,MinR,Min),
ersetzt(Min,EinL,AusL),
ersetzt(Min,EinR,AusR).

Um die rekursive Definition von baum_minbaum2/4 zu erhalten, wird die Regel Falten zweimal angewendet. Hierfür wird die mit Klause K_{10} eingeführte Definition verwendet. Folglich wird Klause K_{15} hinsichtlich der Ziele 1 und 4 unter Verwendung von Klause K_{10} gefaltet und wir erhalten

K_{16}: baum_minbaum2(baum(EinL,EinR),MinL,MinR,baum(AusL,AusR)) ←
baum_minbaum2(EinL,MinL,Min,AusL),
baum_min(EinR,MinR),
wert_wert_min(MinL,MinR,Min),
ersetzt(Min,EinR,AusR).

Nun wird nochmals gefaltet. K_{16} wird hinsichtlich der Ziele 2 und 4 unter Verwendung von K_{10} gefaltet und wir erhalten

K_{17}: baum_minbaum2(baum(EinL,EinR),MinL,MinR,baum(AusL,AusR)) ←
baum_minbaum2(EinL,MinL,Min,AusL),
baum_minbaum2(EinR,MinR,Min,AusR),
wert_wert_min(MinL,MinR,Min).

Dies ist nun die gesuchte rekursive Definition des Prädikates. Um das optimierte

Programm P_{11} zu erhalten, waren somit 11 Transformationsschritte notwendig.

P_{11}:

K_{11}: baum_minbaum(EinBaum,AusBaum) ←
    baum_minbaum2(EinBaum,Min,Min,AusBaum).

K_{14}: baum_minbaum2(blatt(MinL),MinL,MinR,blatt(MinR)).
K_{17}: baum_minbaum2(baum(EinL,EinR),MinL,MinR,baum(AusL,AusR)) ←
    baum_minbaum2(EinL,MinL,Min,AusL),
    baum_minbaum2(EinR,MinR,Min,AusR),
    wert_wert_min(MinL,MinR,Min).

Klause K_{11} ist eine Interface-Regel, um die neue Definition baum_minbaum2/4 in das Programm einzubinden. Programm P_{11} führt die geforderte Wertersetzung durch, indem es den gesamten Baum nur einmal durchwandert. Die Ersetzung der Werte wird nun durch Unifikation durchgeführt. Wird z.B.: die Anfrage ← baum_minbaum(B,MinB) mit

B=baum(blatt(3),baum(blatt(1),blatt(2)))

ausgewertet, so wird die Variable Min, erst nachdem der gesamte Baum durchwandert worden ist, mit 1 unifiziert. Die bis dahin noch nicht belegten Werte in AusBaum sind aber an Min gebunden und werden nun in Folge der Unifikation im gesamten AusBaum gesetzt. D.h., in der optimierten Version wird kein Wert mehr ersetzt, sondern die Unifikation solange aufgeschoben, bis der minimale Wert ermittelt ist.

Die Variable EinBaum tritt in den Prädikaten baum_minbaum/2 und baum_minbaum2/4 nicht mehr in mehreren Zielen auf und konnte als gemeinsame Variable entfernt werden. Dies ist gleichbedeutend mit dem Umstand, daß der Baum nur einmal durchwandert wird und es daher nicht mehr notwendig ist, den Zustand für den zweiten Durchlauf weiterzureichen. Die gemeinsamen Variablen Min, MinL und MinR konnten nicht beseitigt werden.

3.2.3 Ändern der Datenrepräsentation

Listendifferenzen

Der Vorteil in der Verwendung von Listendifferenzen liegt darin, daß das Zusammenfügen von Listendifferenzen konstante Zeit, das Zusammenfügen von normalen Listen aber lineare Zeit - proportional zur Anzahl der Elemente - beansprucht. Es liegt daher nahe, normale Listen durch Listendifferenzen zu ersetzen. Das Problem besteht nun darin, zu einem gegebenen Programm, das normale Listen verwendet, das entsprechende Programm zu finden, das mittels Listendifferenzen formuliert ist. Sei p(X,Y) ein Prädikat in einem Programm *P*. Wir wollen ein Prädikat p_ld(X,L−R) finden, das gilt, wenn p(X,Y) gilt und Y ist repräsentiert durch L−R.

Nehmen wir an, daß das Zusammenfügen von Listen in einem Programm *P* durch ein Prädikat append(X,Y,Z) definiert ist, das gilt, wenn Z jene Liste darstellt, die durch Zusammenfügen von X und Y entsteht. Dann kann die gesuchte Transformation in vielen Fällen mit Hilfe der T&S-Definition Regel unter Verwendung folgender Definition für das Prädikat p_ld/2 gefunden werden:

```
p_ld(X,L−R) ←p(X,Y),append(Y,R,L).
```

Mit Hilfe dieser Definition und der in 3.1 dargestellten Transformationsregeln kann eine rekursive Definition des Prädikates p_ld/2 gefunden werden, die nicht mehr von p/2 oder append/3 abhängig ist. Um p_ld/2 in einem Programm verwenden zu können, bedarf es der Interface-Regel

```
p(X,Y) ←p_ld(X,Y−[]).
```

Gegeben sei das Prädikat liste_sortiert/2, das eine Listen von Elementen sortiert. Das Prädikat partition(Xs,X,Ks0,Gs0) beschreibt zwei Partitionen der Liste Xs, wobei Ks0 alle Elemente kleiner oder gleich X und Gs0 alle Elemente größer als X beinhaltet.

```
liste_sortiert([],[]).
liste_sortiert([X|Xs],Ys) ←
        partition(Xs,X,Ks0,Gs0),
        liste_sortiert(Ks0,Ks),
        liste_sortiert(Gs0,Gs),
        append(Ks,[X|Gs],Ys).
```

Das mittels Listendifferenzen optimierte Programm sieht dann folgendermaßen aus:

```
liste_sortiert(Xs,Ys) ←
        liste_ldsortiert(Xs,Ys−[]).

liste_ldsortiert([],R−R).
liste_ldsortiert([X|Xs],Ys−Zs) ←
        partition(Xs,X,Ks0,Gs0),
        liste_ldsortiert(Ks0,Ys−[X|Ys1]),
        liste_ldsortiert(Gs0,Ys1−Zs),
```

Der Aufruf des Prädikates append/3 konnte beseitigt werden und das Aneinanderhängen der Listen wird nun in konstanter Zeit durchgeführt. Die Darstellung der Listendifferenzen kann nun nochmals durch die in Kapitel 4 dargestellten Techniken optimiert werden.

3.2.4 Rekursionsbeseitigung

Da Prolog keine iterativen Konstrukte besitzt, müssen sämtliche Algorithmen rekursiv implementiert werden. Die häufige Verwendung von Rekursion kann zu Ineffizienz führen, sodaß die Beseitigung der Rekursion, d.h. Umwandlung in eine Iteration, die Effizienz eines Programms erhöhen kann.

Unter Rekursionsbeseitigung versteht man die Herleitung von endrekursiven Klausen. Eine Klause heißt *endrekursiv*, wenn der rekursive Aufruf im Rumpf einer Klause das letzte zu erfüllende Ziel ist; allgemein hat eine endrekursive Klause die Form

$$p(A_1 \dots A_n) \leftarrow \alpha, p(B_1 \dots B_n)$$

wobei $A_1 \dots A_n$ und $B_1 \dots B_n$ Argumente sind und α eine Folge von Literalen.

Die Beseitigung der Rekursion bedarf der Optimierung auf zwei Ebenen. Auf Quelltext-Ebene wird das Programm in ein equivalentes endrekursives Programm transformiert. Durch eine Optimierung auf Ebene des abstrakten Interpreters wird dann die als Endrekursion formulierte Regel iterativ ausgeführt.

Es kann jede rekursive Klause in eine endrekursive transformiert werden, indem die Ziele im Rumpf einer Klause umgeordnet werden. Dadurch kann aber das Maß an Nicht-Determinismus erhöht werden, sodaß die Effizienz des Programms insgesamt nicht gesteigert wird. Weiters kann dadurch das Terminationsverhalten als auch die Reihenfolge der Antwortsubstitutionen verändert werden. Die Semantik von Prolog bleibt im allgemeinen nicht erhalten. Dies ist im folgenden Beispiel zu sehen.

P_0:

```
vorfahre_von(X,X).
vorfahre_von(X,Y) ←
    vorfahre_von(X,P),
    kind_von(Y,P).
```

P_0 terminiert für die Anfrage ← vorfahre_von(X,Y) nicht. Nachdem alle Lösungen gefunden sind, tritt eine Endlosrekursion auf. Werden nun die beiden Ziele vertauscht, so erhält man folgendes Programm:

P_1:

```
vorfahre_von(X,X).
vorfahre_von(X,Y) ←
    kind_von(Y,P),
    vorfahre_von(X,P).
```

Zum Unterschied von P_0 terminiert P_1 für die Anfrage ← vorfahre_von(X,Y). Das Terminationsverhalten wurde durch Vertauschung der Ziele verändert.

Aus diesen Gründen sind mehrere Ansätze entwickelt worden, mittels derer es möglich ist, Endrekursion zu beseitigen, ohne den Nicht-Determinismus zu erhöhen und die Semantik zu erhalten.

Ein Ansatz besteht darin, nahezu-endrekursive Klausen in endrekursive zu transformieren, indem die Regeln Entfalten und Falten angewendet werden. Eine Klause heißt *nahezu−endrekursiv* genau dann, wenn sie der Form

$$p(A_1 \ldots A_n) \leftarrow \alpha, p(B_1 \ldots B_n), \beta.$$

ist, wobei α eine Konjunktion von Zielen ist, und β eine Konjunktion von Zielen, die von p/n nicht abhängig sind. Die Konjunktion β enthält Ziele, die nur Aufrufe von primitiven Zielen aufweisen, wie einfache mathematische Operationen oder das Aneinanderhängen von Listen. Bei der Transformation werden Eigenschaften wie die Assoziativität des Aneinanderhängens von Listen, die Assoziativität und Kommutativität der Addition und die Distributivität der Multiplikation über der Addition ausgenützt [Deb85], [Deb88].

Ein weiterer Ansatz besteht in der Transformation von Programmen in binäre Programme (siehe 4.5.1). Wesentlich ist, daß diese Transformation auf alle Programme anwendbar ist.

Gegeben seien die Klausen

p.
p $\leftarrow$p, q.

Durch Transformation nach binärem Prolog erhalten wir

p $\leftarrow$p(wahr).

p(Kont) $\leftarrow$Kont.
p(Kont) $\leftarrow$p((q,Kont)).

Diese Transformation steigert noch nicht die Effizienz. Erst die Ausführung auf einer spezialisierten Version der WAM, der Binary WAM (BinWAM), zeigt die erwünschte Effizienzsteigerung, da die BinWAM speziell auf die Abarbeitung von Klausen mit nur genau einem Ziel im Rumpf einer Klause optimiert ist. Dies wird noch genauer in 4.5.1 dargestellt.

3.3 Prolog-Semantik

Die Semantik von Prologprogrammen wird in Hinsicht auf Termination und Reihenfolge der Antwortsubstitutionen untersucht. Die Transformationen Falten und Entfalten erhalten die Semantik von Prologprogrammen im allgemeinen nicht. Dies wird durch die folgenden Beispiele verdeutlicht.

P_0: K_1: p(X) $\leftarrow$q(X),r(X).
K_2: q(X) $\leftarrow$fail.
K_3: r(X) $\leftarrow$r(X).

Für die Anfrage $\leftarrow$ p(X) terminiert P_0, da das erste Ziel von $p/1$ immer scheitert. Falten wir nun K_1 hinsichtlich r(X) unter Verwendung von K_1, so erhalten wir Programm P_1:

P_1: K_4: p(X) $\leftarrow$p(X).
K_2: q(X) $\leftarrow$fail.
K_3: r(X) $\leftarrow$r(X).

Im Unterschied zu P_0 terminiert P_1 für die Anfrage $\leftarrow$p(X) nicht und weist somit ein anderes Terminationsverhalten auf. Dies entspricht dem in 3.1.4 erläuterten Selbstfalten.

Da, wie bereits erwähnt, Entfalten das Inverse zu Falten ist, zeigt dieses Beispiel auch das Ändern der Terminationseigenschaft bei Entfalten. Wird in P_1 die Klause K_4 unter Verwendung der Klause K_1 ($\in P_0$) entfaltet, so erhalten wir wieder die Klause K_1. Das durch diesen Transformationsschritt erhaltene Programm P_2 ist equivalent mit Programm P_0 und hat daher ein anderes Terminationsverhalten als P_1.

Die Reihenfolge der Antwortsubstitutionen kann ebenfalls verändert werden, wie das folgende Beispiel zeigt.

P_0: K_1: p(X,Y) ←q(X),r(Y).
K_2: q(a).
K_3: q(b).
K_4: r(a).
K_5: r(b).

Die Antwortsubstitutionen für ←p(X,Y) sind {X/a,Y/a}, {X/a,Y/b}, {X/b,Y/a}, {X/b,Y/b}. Nach entfalten von K_1 hinsichtlich r(Y) erhalten wir

P_1: K_6: p(X,a) ←q(X).
K_7: p(X,b) ←q(X).
K_2: q(a).
K_3: q(b).

Die Antwortsubstitutionen für ←p(X,Y) sind {X/a,Y/a}, {X/b,Y/a}, {X/a,Y/b}, {X/b,Y/b}. Sie unterscheiden sich in der Reihenfolge durch die Substitutionen 2 und 3 für die Variablen X und Y. Die unterschiedliche Reihenfolge wird verursacht durch die links-nach-rechts Auswertungsreihenfolge von Prolog. Wahlpunkte werden in Prolog immer dann erzeugt, wenn für eine Zielreduktion mehr als eine Klause dem Interpreter zur Auswahl steht. In obigem Beispiel gibt es für die Reduktion von q(X) zwei Klausen, nämlich $q(a)$ und $q(b)$. Immer wenn eine derartige Wahlmöglichkeit im Laufe eines Beweises auftritt, wird ein sogenannter Wahlpunkt - eine interne Datenstruktur - angelegt, um bei Backtracking und Aufheben einer bestehenden Variablenbindung andere mögliche Substitutionen durchführen zu können. Die unterschiedliche Reihenfolge ist also eine Folge des Backtrackings und der veränderten Reihenfolge der Wahlpunkte.

Folgerung. Die Reihenfolge der Antwortsubstitutionen zweier Programme P_k und des durch eine Transformation erhaltenen Programms P_{k+1} ist genau dann gleich, wenn die Reihenfolge der während der Interpretation des Programms generierten Wahlpunkte gleich ist und die Lösungsmengen übereinstimmen.

Zusammenfassung

In diesem Kapitel wurden Methoden für die Programmoptimierung vorgestellt. Wir konnten Zwischenvariablen und redundante Berechnungen beseitigen. Die Struktur rekursiver Programme wurde verändert, sodaß Datenstrukturen weniger oft durchwandert werden mußten. Die vorgestellten Strategien waren das Tupeln, die Schleifenabsorption und die Generalisation. Allen drei Strategien war gemeinsam, daß ein neues Prädikat eingeführt werden mußte. Dieses Prädikat wurde Eureka-Prädikat genannt. Durch Entfalten und Falten wurde dann ein neues effizienteres Programm hergeleitet.

Weiters haben wir Strategien kennengelernt, die auf diesen drei grundlegenden Strategien aufbauen. So konnte etwa durch das Ändern der Datensrepräsentation das Programm optimiert werden. In Prologprogrammen wurden normale Listen durch Listendifferenzen ersetzt. Eine weitere Methode zur Effizienzsteigerung war das Überführen von rekursiven in iterative Programme. Dabei nutzten wir die Optimierung des letzten Aufrufs einer Klause auf WAM-Ebene. Abschließend wurden die Auswirkungen der Transformationen auf die Semantik von Prologprogrammen untersucht. Dabei stellte sich heraus, daß das Selbstfalten von Klausen die Terminationseigenschaft von Programmen ändert und daher bei der Optimierung von Programmen vermieden werden muß.

Allerdings konnten nur einige der gemeinsamen Variablen in Prädikaten optimiert werden. Im folgenden Kapitel werden Kontinuationstransformationen eingehend untersucht. Durch diese wird es möglich sein, auch diese Variablen zu optimieren.

Kapitel 4

Kontinuationstransformationen

In diesem Kapitel stellen wir eine Transformation vor, mit der unnötige Variablen beseitigt werden können. Es werden existentielle Variablen und redundante Strukturen, wie sie bei der Verwendung von Listendifferenzen und Akkumulatoren auftreten, beseitigt. Die Transformationen haben eine direkte Auswirkung auf die Registerallokation der WAM. Die Lebensdauer von WAM-Registern wird erhöht, sodaß Werte länger im selben Register gehalten werden können. Alle Transformationen beruhen auf lokalen Optimierungen. Es werden keine globalen Datenflußanalysen durchgeführt.

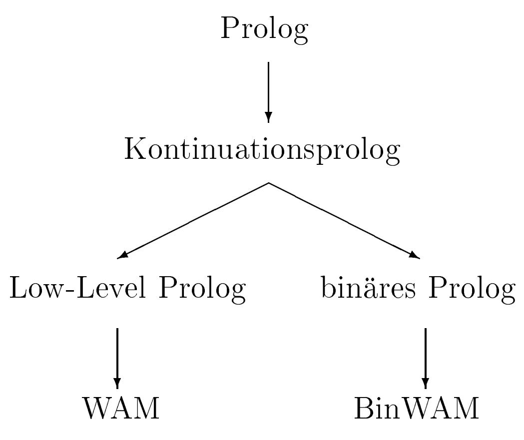

Abbildung 4.1: Die Phasen des Transformationsprozesses

Der Transformationsprozeß ist in Abbildung 4.1 dargestellt. Bevor das Programm optimiert werden kann, wird es nach Kontinuationsprolog übersetzt. Dies ist eine für die hier vorgestellten Optimierungen besonders gut geeignete Darstellung. Sämtliche

Optimierungen werden in Kontinuationsprolog durchgeführt. Alle Optimierungsschritte erhalten die Semantik (siehe 3.3 auf Seite 31) der Programme. Das optimierte Programm kann dann nach verschiedenen abstrakten Maschinen übersetzt werden. Die hier vorgestellten Transformationen beschreiben die Übersetzung nach Low-Level Prolog und binärem Prolog. Die Optimierungen sind unabhängig von der Implementation und erst die Übersetzung des optimierten Programms nach der jeweiligen abstrakten Maschine muß an diese angepaßt werden. Low-Level Prolog ist Prolog, das um built-in Prädikate erweitert wurde. Die Übersetzung von Low-Level Prolog in WAM-Code schließt den Transformationsprozeß ab. Jedes Kontinuationsprologprogramm ist direkt nach binärem Prolog übersetzbar. Binäres Prolog ist eine spezielle Form von Prolog, das für die Ausführung auf einer BinWAM am besten geeignet ist. Diese abstrakte Maschine ist eine spezialisierte Variante der WAM, die nur binäre Klausen verarbeitet.

Die Transformationen wurden ursprünglich in binärem Prolog durchgeführt. Erst später wurde die Zwischendarstellung Kontinuationsprolog entwickelt. Die in Kapitel 3 erläuterten Transformationen waren Ausgangspunkt für die Entwicklung von Kontinuationstransformationen, durch die auch Differenzvariablen beseitigt werden. So war 1992 zu lesen:

> Die Entwicklung von EBC-Transformationen wurde durch die ursprünglichen Probleme von Programmtransformationen motiviert. Mit deren Hilfe werden unnötige Variablen beseitigt, um das Falten von Klausen zu ermöglichen. Während der Entwicklung von EBC-Transformationen stellte sich auch heraus, daß sie eine interessante Optimierungstechnik darstellen, da sie die Größe von Environments und Kontinuationen verkleinern [Neu92a].

Der Name EBC-Transformationen wurde später in Kontinuationstransformationen abgeändert [Neu97]. Dem Begriff der unnötigen Variable entsprechen die Begriffe Kontext- und Differenzvariable. Die Verkleinerung eines Environments wird durch die Verringerung von permanenten Variablen (siehe 5.2) in einer Klause erreicht. Der wesentliche Unterschied gegenüber den Techniken Falten/Entfalten liegt darin, daß Differenzvariablen durch Falten/Entfalten nicht beseitigt werden können [Neu92a].

Ziel dieser Transformationen ist es, eine verbesserte Registerallokation bei Ausführung des WAM-Codes (bzw. BinWAM) zu erreichen, wenngleich die Optimierungen ausschließlich durch Transformation der Kontinuationsprologklausen erreicht werden. Die höhere Effizienz des erzeugten WAM-Codes ist eine Folge der Beseitigung von Redundanz im Prologprogramm, sodaß der WAM-Code weniger Befehle aufweist, als der, der für das ursprüngliche Programm generiert wird. Auf diesen letzten Schritt wird noch ausführlich in 4.5 eingegangen.

4.1 Optimierbare Programmkonstrukte

Die im Folgenden vorgestellten Optimierungen sind nur für gewisse Datenstrukturen anwendbar. Allen Techniken ist gemeinsam, daß sie einen Zustand manipulieren, der explizit weitergereicht wird. Durch die Anwendung verschiedener Transformationsregeln wird die Redundanz dieser Datenstrukturen explizit gemacht.

Existenzvariablen (bzw. Zwischenvariablen) sind Variablen, die nur im Rumpf einer Klause vorkommen. In der Klause p(X0,X) $\leftarrow$q(X0,$\underline{\text{X1}}$),q($\underline{\text{X1}}$,X). ist X1 eine existentielle Variable.

Differenzen werden durch Paare von Variablen dargestellt. Diese Datenstruktur wird ausführlich auf Seite 11 erläutert. Beim Durchreichen erzeugt das Differenzpaar eine Existenzvariable, sodaß die Optimierung von Differenzen auch das Beseitigen einer oder mehrerer Existenzvariablen miteinschließt. Der zweistellige Funktor '–', den wir in Kapitel 2 benutzt hatten, wird im Folgenden weggelassen. Stattdessen wird für jede der beiden Variablen ein Argument verwendet. D.h. statt p(A–B) verwenden wir p(A,B).

Kontextargumente werden verwendet, um eine 1-n Beziehung auszudrücken, d.h. wenn ein einzelnes Element mit mehreren anderen in Beziehung steht. Wie noch gezeigt wird, ist in einigen Fällen auch das Einführen von zusätzlichen Kontextargumenten sinnvoll, um Optimierungen anwenden zu können.

Die Übersetzung des Prologprogramms nach Kontinuationsprolog steht am Anfang des Optimierungsprozesses. Die hier kurz erläuterten Datenstrukturen werden dadurch nicht verändert. Erst die Anwendung der Transformationen macht Redundanz sichtbar, deren Beseitigung eine verbesserte Registerallokation auf WAM-Ebene zur Folge hat.

4.2 Kontinuationsprolog

Kontinuationsprolog ist eine Zwischensprache in der Übersetzung von Prolog nach der abstrakten Maschine. Optimierungen, die unter Verwendung von Kontinuationsprolog durchgeführt werden, sind unabhängig von der Implementation, da nach Optimierung des Programms, eine Übersetzung auf verschiedene abstrakte Maschinen möglich ist.

Ein Kontinuationsprologprogramm unterscheidet sich von einem Prologprogramm im Wesentlichen darin, daß im Kopf einer Klause mehrere Elemente auftreten können. Eine Kontinuationsprologklause hat also die Form

$$\alpha \twoheadleftarrow \beta.$$

wobei α und β Folgen von Kontinuationselementen sind. Solche Folgen heißen *Kontinuationen*. Kontinuationsprolog ist eine Obermenge von Prolog. Um Transformationen in Kontinuationsprolog durchführen zu können, muß das Prologprogramm nach Kontinuationsprolog übersetzt werden. Dieser Vorgang wird im Folgenden definiert.

Definition. Sei A der Kopf einer Prologklause und β der Rumpf einer Prologklause bestehend aus beliebig vielen Zielen. Man erhält eine Kontinuationsprologprogramm, indem alle Klausen eines Programms P derart abgebildet werden, daß der Kopf auf eine einelementige Kontinuation und der Rumpf auf eine Folge von Kontinuationselementen abgebildet wird. Wir erhalten also

$$[A] \twoheadleftarrow \beta.$$

D.h., nachdem ein Prologprogramm nach Kontinuationsprolog übersetzt worden ist, enthält jeder Kopf nur ein Element. Ein Kopf mit mehreren Elementen entsteht durch die Anwendung von Transformationsregeln. Für die Übersetzung von Kontinuationsprolog nach einer abtrakten Maschine muß eine Entsprechung für das mehrfachen Auftreten von Elementen im Kopf einer Klause gefunden werden. Dies wird ausführlich in 4.5 erläutert. Die Falten/Entfalten-Systeme, die in 3.2 dargestellt wurden, sind nicht anwendbar, da sie Relationen auf der Ebene von Zielen definieren [Neu97].

4.3 Regeln

Kontinuationstransformationen bestehen wie die in Kapitel 3 beschriebenen Transformationen aus Regeln und Strategien. Die Regeln werden iterativ angewendet, bis die erwünschte Optimierung des Programms erreicht ist. Der Transformationsprozeß beginnt mit Einführen von Gleichungen (R1), welche dann in das Programm compiliert werden (R2,R3). Daraus ergeben sich meist Konstellationen, in denen die Kontinuation verkürzt (R4) oder Argumente verallgemeinert werden können (R5). Wenn nötig, wird nun wieder mit der Anwendung von Regel R1 fortgefahren. In der Iteration werden die Regeln R1 und R2 jeweils einmal angewendet, die Regeln R3 bis R5 wenn nötig mehrmals.

Das optimierte Kontinuationsprologprogramm kann dann nach verschiedenen abstrakten Maschinen übersetzt werden. Die hier beschriebenen Maschinen sind die WAM (Warren Abstract Machine) und die BinWAM, eine spezialisierte Variante der WAM.

4.3.1 Regel R1: Gleichungseinführung

Gemäß den Gleichungen, die unter Verwendung dieser Regel eingeführt werden, werden Terme ersetzt. Ziel dieser Ersetzungen ist es, vorhandene Redundanz explizit zu machen und zu beseitigen. Alle Gleichungen, die einem Programm hinzugefügt werden, müssen gewissen Bedingungen genügen, sodaß die Semantik des ursprünglichen Programms durch Anwendung der Gleichungen bzw. Unifikation nicht verändert werden kann. Diese Bedingungen beschreiben auch implizit die Teilmenge von Prologprogrammen, die mit Hilfe dieser Gleichungen optimiert werden kann.

Definition. Gegeben sei ein Programm P_k. Wir erhalten ein Programm P_{k+1} indem eine Menge von Kontinuationsgleichungen der Form $s \doteq t$ hinzugefügt wird. Eine Kontinuationsgleichung muß folgenden Bedingungen genügen:

- Jeder Kontinuationsfunktor, der nicht in P_k auftritt, ist das erste Element der Kontinuation.
- Treten in zwei Gleichungen die gleichen Kontinuationsfunktoren an der gleichen Stelle auf, so müssen an darauffolgender Stelle verschiedene Funktoren auftreten.

Seien a_i alte und n ein neuer Kontinuationsfunktor. Die Gleichung $[a_1,a_2] \doteq [n]$ ist zulässig. Hingegen ist $[a_1,a_2] \doteq [a_i,n]$ nicht zulässig, da der neue Kontinuationsfunktor n nicht als erstes Element auftritt. Betrachten wir nun eine Menge von Gleichungen. Die Menge von Gleichungen $\{[a] \doteq [n,a_1]\ ,\ [a] \doteq [n,a_2]\ \}$ ist zulässig. Die Menge von Gleichungen $\{[a] \doteq [n,a_1]\ ,[a] \doteq [n,a_1]\ \}$ ist nicht zulässig, da in beiden Gleichungen an zweiter Stelle der gleiche Funktor auftritt, obwohl beide an erster Stelle einen neuen Funktor aufweisen und somit die zweite Bedingung nicht erfüllt ist. Die Gleichung $[a_1,a_2] \doteq [a_1,n,a_2]$ [1] ist nicht zulässig, da der neue Kontinuationsfunktor nicht an erster Stelle auftritt.

Um diese Gleichungen in das Programm zu compilieren, kommen die nun folgenden Regeln R2 und R3 zur Anwendung, die Bedingungen für das Compilieren des Kopfs und des Rumpfs einer Klause festlegen.

4.3.2 Regel R2: Kopf-Compilierung

Gleichungen werden in das Programm compiliert, indem alle unifizierbaren Köpfe ersetzt werden. Es wird sichergestellt, daß alle Arten von alternativen Repräsentationen für eine Kontinuation vom Kopf gelesen werden können. Dies ist notwendig, da

[1] Diese generische Gleichung ist in [Neu93a] angeführt und als konservative Erweiterung zulässig. Die hier verwendeten Bedingungen sind jedoch strenger und schreiben die Verwendung von neuen Kontinuationsfunktoren an erster Stelle vor. Alle in [Neu93a] durchgeführten Transformationen sind mit dieser neuen Definition aus [Neu97] ebenso durchführbar.

der Kopf die Kontinuation liest und unifiziert. Bei der Anwendung der Gleichungen ist keine der beiden Seiten der Gleichung ausgezeichnet, d.h., daß die Gleichungen in beiden Richtungen verwendet werden können.

Definition. Sei P_k ein Kontinuationsprologprogramm bestehend aus Klausen K_i der Form $r \cdot s \cdot t \cdot \twoheadleftarrow z$ und Gleichungen E_j der Form $u \cdot v \doteqdot w$.

Für eine Gleichung E_j sind folgende Ersetzungen möglich:

1. $(r \cdot w \cdot t \twoheadleftarrow z)\theta$ wenn $\exists\theta.s\theta = (u \cdot v)\theta$.

2. $(r \cdot s \cdot w \twoheadleftarrow z \cdot v)\theta$ wenn $\exists\theta.t\theta = u\theta$.

Wir erhalten ein Programm P_{k+1} indem für jede Klause K_i und Gleichung E_j die Klause, in der Ersetzungen durchgeführt wurden, hinzugefügt wird.

Ersetzungsregel 1 wird angewendet, wenn eine Seite einer Gleichung mit dem Kopf einer Klause unifizierbar ist. Ersetzungsregel 2 kommt zur Anwendung, wenn die Seite einer Gleichung mit dem Kopf überlappt. D.h. es ist nur ein Teil der Gleichung, nämlich u, mit dem Kopf einer Gleichung unifizierbar. In manchen Fällen sind aber auch beide Ersetzungsregeln anwendbar. Dies ist immer dann der Fall, wenn einer der Ausdrücke r,s,t,v oder w eine leere Kontinuation ist. Ist z.B. w leer, so wird die Klause $u \cdot v \twoheadleftarrow []$ dem Programm hinzugefügt.

Beispiel 1: Sei P_0 $\langle [p(X)] \twoheadleftarrow [q(X)] \rangle$ und E die Gleichung $[p(X)] \doteqdot [r(X)]$. In diesem Falle sind beide Ersetzungsregeln anwendbar.

Unter Anwendung von Ersetzungsregel 1 werden die Ersetzungen folgendermaßen durchgeführt:

> Klause: $[p(X)] \twoheadleftarrow [q(X)]$ der Form $r \cdot s \cdot t \cdot \twoheadleftarrow z$
> also ist $r = \epsilon$, $s = p(X)$, $t = \epsilon$ und $z = q(X)$.
>
> Gleichung: [p(X)] $\doteqdot$ [r(X)] der Form $u \cdot v \doteqdot w$
> also ist $u = p(X)$, $v = \epsilon$ und $w = r(X)$.
>
> Bedingung: $(u \cdot v)\theta = t\theta$ ist erfüllt.
>
> Die Ersetzung unter Verwendung von Ersetzungsregel 1 ist $(r \cdot w \cdot t \twoheadleftarrow z)\theta$.
> Also wird die Klause $[r(X)] \twoheadleftarrow [q(X)]$ dem Programm hinzugefügt.

Das gleiche Ergebnis ist in diesem Fall auch mit Ersetzungsregel 2 zu erhalten. Die Ersetzungen werden folgendermaßen durchgeführt:

Klause: $[p(X)] \twoheadleftarrow [q(X)]$ der Form $r \cdot s \cdot t \cdot \twoheadleftarrow z$
also $r = \epsilon$, $s = \epsilon$, $t = p(X)$ und $z = q(X)$.

Gleichung: [p(X)] $\doteq$ [r(X)] der Form $u \cdot v \doteq w$
also $u = p(X)$, $v = \epsilon$ und $w = r(X)$

Bedingung: $u\theta = t\theta$ ist erfüllt.

Ersetzungsregel 2: $(r \cdot s \cdot w \twoheadleftarrow z \cdot v)\theta$
also erhalten wir ebenso mit Regel 2: $[r(X)] \twoheadleftarrow [q(X)]$.

Beispiel 2: Sei P_0 $\langle [p(X), q(X)] \twoheadleftarrow [] \rangle$ und E die Gleichung $[q(X), r(X)] \doteq [s(X)]$. Unter Anwendung von Ersetzungsregel 2 wird die neue Klause $[p(X), s(X)] \doteq [r(X)]$ dem Programm hinzugefügt. Regel 1 ist hier nicht anwendbar, da nicht die gesamte linke Seite der Gleichung mit einem Element des Kopfs der Klause unifizierbar ist.

Definition. Sei P_k ein Programm und E ein Menge von Gleichungen. Die Funktion $Orig(K)$ berechnet die Menge der ursprünglichen Klausen, die, aufgebaut aus den Symbolen aus P_0, unter Verwendung von E in K transformiert werden können.

$Orig$ berechnet also jene Klausen, die durch Verwendung von E ausgehend von K entstehen können, wenn man ausschließlich Symbole aus P_0 verwendet. Nur eine Teilmenge dieser Klausen sind jene, die man durch Anwendung der Gleichung E auf Klausen des Programms P_0 erhält. Die obige Definition beschreibt eine Obermenge dieser Klausen, da durch $Orig$ auch jene berechnet werden, die nicht als Klausen in P_0 enthalten waren, sondern die aus Symbolen dieser aufgebaut sind und daher auch andere Form haben können.

Beispiel: Gegeben sei die Klause [a] und die Gleichung [a] $\doteq$ [b(X)] . Dann kann unter Verwendung der Gleichung die Klause [b(X)] erzeugt werden, aber auch, da a ein Symbol aus P_0 ist, die Klause [b(a)].

Definition. Die Funktion p_i definiert das Präfix einer Kontinuation mit n Elementen bis zum i-ten Element, mit $0 \leq i \leq n$.

Das Präfix ist also eine Folge von i Kontinuationselementen. Sei z.B. die Kontinuation [e1,e2] gegeben, dann ist p_0([e1,e2])=[], p_1([e1,e2])=[e1] und p_2([e1,e2])=[e1,e2].

Definition. Seien K und K_g Klausen der Form $A \twoheadleftarrow B$ und $A_g \twoheadleftarrow B_g$. K und K_g heißen *scankompatibel*, wenn für alle i,$j \geq 0$ gilt:

$$Orig(p_i(H) \twoheadleftarrow p_j(B)) = Orig(p_i(H_g) \twoheadleftarrow p_j(B_g))$$

Diese Bedingung stellt sicher, daß während der Programmausführung alle Variablenbindungen der äußeren Kontinuationen zu denen des ursprünglichen Programms identisch sind. D.h., daß auch built-in Prädikate, cuts, etc. in den zu transformierenden Programmen an beliebiger Stelle verwendet werden können. Es wird sichergestellt, daß die gleichen Variablenbindungen durchgeführt werden, wie im ursprünglichen Programm, da sonst die Gleichungen nicht gültig wären.

4.3.3 Regel R3: Zielersetzung

Die Klause K der Form $q \twoheadleftarrow r \cdot s \cdot t$ kann durch die Klause D der Form $r \cdot v\theta \cdot t$ unter Verwendung der Gleichung $u \doteq v$, genannt E, ersetzt werden, wenn die Bedingungen

1. $(q \twoheadleftarrow r \cdot t)\theta = q \twoheadleftarrow r \cdot t$
2. $Orig(K) = Orig(D)$

erfüllt sind.

Bedingung 1 sichert zu, daß die bei der Transformation verwendete Substitution θ keine Substitutionen im Kopf oder anderen Teilen des Rumpfs bewirkt, als jenem, der durch die Gleichung E transformiert wird. Bedingung 2 sichert zu, daß die Klause D nur aus Symbolen besteht, die entweder im ursprünglichen Programm enthalten waren, oder durch Gleichungen eingeführt worden sind. Man beachte, daß auch das Einführen neuer Symbole mittels Gleichungen gewissen Bedingungen genügen muß (siehe 4.3.1 auf Seite 38).

Die folgenden beiden Regeln R4 und R5 dienen dem Beseitigen von Redundanz in einem Programm. Die Bedingungen für diese beiden Regeln sind ausschließlich von Gleichungen abhängig, die mittels der Regeln R1-R3 zuvor in das Programm compiliert worden sind. Deswegen wird auch von einer Iteration der Regeln R1-R5 gesprochen und die Anwendung der Regeln R1-R5 als ein Schritt in der Transformation bezeichnet, da einige Bedingungen nur von der Anwendung anderer Regeln in der jeweiligen Iteration abhängig sind. Während Regel R4 redundante Ausdrücke entfernt, ist Regel R5, die Argumentgeneralisation, oft eine Vorbereitung zur Anwendung von Regel R4.

4.3.4 Regel R4: Kontinuationskürzung

Eine Klause der Form $H \cdot X \twoheadleftarrow B \cdot X$ kann gekürzt werden zu $H \twoheadleftarrow B$, wenn

1. die Variablen in X nicht in H oder B vorkommen und
2. $Orig(H \cdot X \twoheadleftarrow B \cdot X) = Orig(H \twoheadleftarrow B)$.

Bedingung 2 sichert zu, daß X nur durch Anwendung von Gleichungen erzeugt worden ist. Dies tritt insbesondere dort auf, wo durch Einführen von Gleichungen Redundanz im Programm explizit gemacht wird. Durch Anwendung der Regel R4 wird diese Redundanz beseitigt. Das folgende Beispiel zeigt, wann Regel 1 das Kürzen der Kontinuation verhindert.

[p(s(El),El),rest(El)] $\twoheadleftarrow$
    [p(s,El),
    rest(El),
    p(s,El),
    rest(El)].

Somit ist $H =$ p(s(El),El) und $B =$ p(s,El),rest(El),p(s,El). X könnte man mit rest(El) gleichsetzen. Regel 1 verhindert aber, daß das Kontinuationselement rest(El) gekürzt wird, da die Variable El auch in B und K vorkommt und Regel R4 somit nicht anwendbar ist. Dennoch ist es möglich, die Kontinuation auch in diesem Falle zu kürzen. Es muß ein Programm erzeugt werden, das Bedingung 1 erfüllt, aber die Semantik des Programms nicht verändert. Dies motiviert die Definition einer Generalisation, die die Umbenennung der Variablen erlaubt, sodaß die Variable El nur mehr in der zu kürzenden Kontinuation vorkommt.

4.3.5 Regel R5: Argumentgeneralisation

Die Klause K kann durch die Klause $K_g\theta$ ersetzt werden, wenn

1. die Variablen in $dom(\theta)$ in K_g nur als Subterme der Kontinuationselemente auftreten,
2. $dom(\theta) \subseteq vars(B_g)$ und
3. K mit K_g scankompatibel ist.

Als Beispiel einer Anwendung der Generalisation sei hier nochmals das obige Prädikat p/2 betrachtet. Die Variable El kommt nicht nur in der zu kürzenden Kontinuation vor, sondern auch in anderen Elementen. Um eine Kürzung der Kontinuation durchführen zu können, d.h. mittels Regel R4 Redundanz beseitigen zu können, müssen die Vorkommnisse der Variable El als verschiedene Variablen auftreten. Daher wird obige Klause mittels Regel R5 verallgemeinert, sodaß wir folgende Klause erhalten:

[p(s(El),El),~~rest(El2)~~] $\twoheadleftarrow$
    [p(s,El1),
    rest(El1),
    p(s,El2),
    ~~rest(El2)~~].

Nun kann rest(El2) mittels Regel R4 gekürzt und Redundanz im Programm vermindert werden.

Das folgende minimale Beispiel veranschaulicht die Anwendung der fünf Regeln auf ein Variablenpaar.

```
[varpaar(X,X)] ↞
        [].
[varpaar(X0,X)] ↞
        [varpaar(X0,X1),
        varpaar(X1,X)].
```

Diese Prädikat hat eine unendliche Lösungssequenz gleicher Terme, wenn das erste mit dem zweiten Argument unifizierbar ist. Anderenfalls terminiert es nicht. Das Prädikat stellt die Struktur eines Differenzpaares dar, wenngleich hier die Differenz nicht explizit sichtbar ist. Diese wird erst durch die folgenden Transformationsschritte explizit gemacht. Das Variablenpaar wird aufgespalten und durch Einführen neuer Kontinuationsfunktoren werden redundante Elemente sichtbar gemacht. Diese sind syntaktisch gleich.

Die Variable X kommt sowohl im Kopf als auch im Rumpf der zweiten Klause vor. Die Variable wird also vom Kopf gelesen, um wieder im Rumpf geschrieben zu werden. Die Anwendung der Regeln R1-R5 erzeugt ein Programm, dessen Übersetzung nach Low-Level Prolog oder binärem Prolog ein equivalentes Verhalten wie das ursprüngliche Programm zeigt, aber dessen Ausführung effizienter ist. Der Verminderung an Redundanz im Kontinuationsprologprogramm entspricht die Verminderung von Befehlen im WAM-Code.

Wir beginnen mit Anwendung von Regel R1, also mit Einführen einer Gleichung E_1: [varpaar(X0,X)] $\doteq$ [varpaar2(X0),rest(X)] . Es wird das Argumentepaar (X0,X) aufgespalten, um die Redundanz im Programm explizit zu machen bzw. syntaktisch gleiche Kontinuationselemente zu erzeugen.

Durch Anwendung von Regel R2, d.h. Compilierung der Gleichung in den Kopf der Klause, erhalten wir die Klausen

```
[varpaar2(X),rest(X)] ↞
        [].
[varpaar2(X0),rest(X)] ↞
        [varpaar(X0,X1),
        varpaar(X1,X)].
```

und durch zweimalige Anwendung von Regel R3, d.h. Compilierung der Gleichung in den Rumpf der Klause erhalten wir

[varpaar2(X0),~~rest(X)~~] $\twoheadleftarrow$
    [varpaar2(X0),
    rest(X1),
    varpaar2(X1),
    ~~rest(X)~~].

In dieser Klause kommt das Kontinuationselement rest(X) sowohl im Kopf als auch im Rumpf als letztes Element vor. Dies ist die redundante Datenstruktur, die nun explizit gemacht ist und beseitigt werden kann. Gemäß Regel R4 können die Kontinuationen der zweiten Klause um dieses Element gekürzt werden und wir erhalten

[varpaar2(X0)] $\twoheadleftarrow$
    [varpaar2(X0),
    rest(X1),
    varpaar2(X1)].

Das Kontinuationselement rest(X) kommt nicht mehr im Kopf vor, und wird daher nicht mehr gelesen. Die Redundanz ist beseitigt worden. Es wurde ein Argumentregister, nämlich X, eingespart.

An dieser Stelle kann das Programm bereits direkt nach Low-Level Prolog übersetzt werden. Dieser Transformationsschritt ist ausführlich in 4.5.3 dargestellt.

Für die Übersetzung nach binärem Prolog müssen weitere Transformationsschritte durchgeführt werden, die das Einführen eines Hilfsprädikates miteinschließen. Die Übersetzung nach binärem Prolog ist an sich immer möglich, da sämtliche Transformationsschritte auch in binärem Prolog durchführbar sind. Allerdings ist der hier vorgestellte Formalismus leichter implementierbar und die einzelnen Schritte werden übersichtlicher dargestellt.

4.4 Strategien

Die hier vorgestellten Strategien manipulieren verschiedene Arten von Variablen. Es können Kontextvariablen, Differenzvariablen und existentielle Variablen optimiert werden (siehe 4.1 auf Seite 36). Kontextvariablen werden verwendet, um Werte über mehrere Ziele hinweg durchzureichen. Es wird also die gleiche Variable in mehreren Zielen verwendet. Durch Generalisation erhält man ein Programm, das Variablenpaare enthält, wie sie auch bei Listendifferenzen auftreten. Listendifferenzen entsprechen der im minimalen Beispiel varpaar/2 dargestellten Form. Bei Listendifferenzen treten einige Variablen nur im Rumpf einer Klause auf. Diese heißen Existenzvariablen. Im minimalen Beispiel varpaar/2 ist die Variable X1 eine existentielle Variable.

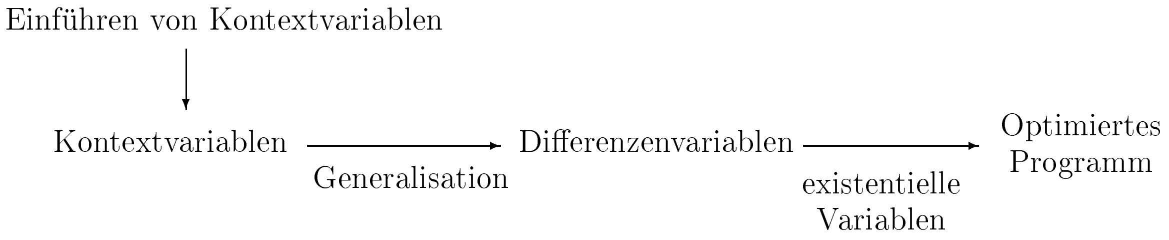

Abbildung 4.2: Transformationsstrategien

In Abbildung 4.2 ist der Zusammenhang zwischen den einzelnen Strategien dargestellt. Pfeile stehen für das Anwenden von Transformationsregeln. Kontextvariablen werden eingeführt, um ein Programm zu erhalten, das optimierbar ist. Kontextvariablen werden generalisiert, damit sie wie Differenzvariablen optimiert werden können. Bei der Verwendung von Differenzvariablen treten im Rumpf von Klausen Existenzvariablen auf. Diese werden durch den Transformationsprozeß beseitigt.

4.4.1 Einführen von Kontextargumenten

In vielen Fällen sind weder Kontextargumente noch Differenzen in Programmen zu finden. Es ist aber möglich, durch Einführen von Kontextargumenten Programme in solche überzuführen, daß sie mittels der im Folgenden beschriebenen Strategien optimiert werden können. Ein Kontextargument wird in mehreren Zielen einer Klause verwendet, um eine 1-n Beziehung auszudrücken. Die Optimierung von Kontextargumenten ist nur dann anzuwenden, wenn jedes der Ziele zwischen den einzelnen Zielen, in denen das zu optimierende Argument auftritt, die Kontextvariable enthält. Gehen wir aus von einem Programm folgender Gestalt:

$$
\begin{array}{l}
[\mathrm{p}(A_1 \ldots A_n)] \twoheadleftarrow \\
\qquad [q_1(A_i,\ldots,\mathrm{K},\ldots,A_{n_1}), \\
\qquad q_2(A_j,\ldots,A_{n_2}), \\
\qquad \vdots \\
\qquad q_m(A_k,\ldots,\mathrm{K},\ldots,A_{n_o})]. \\
\qquad \vdots
\end{array}
$$

mit i,j,k beliebig.

Dann kann durch Einführen eines Kontextarguments in den Zielen q_2 bis q_{m-1} ein Programm erhalten werden, bei dem die Strategie für Optimierung von Kontextargumenten anwendbar ist.

Erweitern wir also obiges Programm um die Kontextvariable K, so erhalten wir:

$$
\begin{array}{l}
[\mathrm{p}(A_1 \ldots A_n)] \twoheadleftarrow \\
\qquad [q_1(A_i,\ldots,\mathbf{K},\ldots,A_{n_1}), \\
\qquad q_2(A_j,\ldots,\mathbf{K},\ldots,A_{n_2+1}), \\
\qquad \vdots, \\
\qquad q_m(A_k,\ldots,\mathbf{K},\ldots,A_{n_o})]. \\
\qquad \vdots
\end{array}
$$

mit i,j,k beliebig.

Die Ziele q_l, mit $2 \leq l \leq m-1$, werden um ein Argument erweitert. Diese müssen daher im gesamten Programm durch die neue Repräsentation ersetzt werden. Die Variable K kann nun als Kontextvariable optimiert werden. Nach Optimierung des Kontextarguments muß der Wert der Variable K nicht mehr in einem Environment (siehe 5.7) gespeichert werden. Wir zeigen dies an Hand eines Beispiels. Gegeben sei das Prädikat nachfahre_von/2 und beliebig viele Fakten kind_von/2:

nachfahre_von(X,X).	$\Longrightarrow$	nachfahre_von(X,X).
nachfahre_von(X,Y) ←		nachfahre_von(**X**,Y) ←
kind_von(P,Y),		kind_von2(**X**,P,Y),
nachfahre_von(X,P)		nachfahre_von(**X**,P)
kind_von(b,a).		kind_von2(_,b,a).
kind_von(c,b).		kind_von2(_,c,b).
⋮		⋮

In nachfahre_von/2 ist die Variable X nicht optimierbar. Wir erweitern das Prädikat kind_von/2 um ein Argument und erhalten kind_von2/3. Das neue Argument nimmt die Variable X auf, die zum zweiten Ziel durchgereicht wird. Wir haben also ein Kontextargument eingeführt. Die Variable X kann nun durch Kontinuationstransformationen optimiert werden. Es wurde die erste Stelle für das Kontextargument gewählt, damit nach der Transformation der Wert von X auf WAM-Ebene im selben Argumentregister gehalten werden kann. Der Wert des ersten Arguments wird im Prädikat kind_von2/3 dann nicht verändert. Es werden keine WAM-Befehle für dieses Argument generiert, sodaß X einfach durchgereicht wird und für den rekursiven Aufruf im ersten Argumentregister bereit steht. Weiters kann die Variable P optimiert werden. Wie Kontextvariablen optimiert werden, erläutern wir an Hand eines Beispiels im folgenden Abschnitt.

4.4.2 Optimierung von Kontextargumenten

Kontextargumente sind ein Konstrukt, das des öfteren in Prologprogrammen Verwendung findet. Sie werden verwendet, wenn ein einzelnes Element mit mehreren anderen in Beziehung steht. In dem folgenden Beispiel wird unter anderem auch die Verwendung von Argumentgeneralisation (Regel R5) demonstriert. Als Beispiel verwenden wir das Prädikat baum_element/2. Es beschreibt Bäume, deren Knoten nur das Element El enthalten. Die Kontextvariable ist El und wird in beiden rekursiven Aufrufen verwendet.

```
baum_element(nil,_El).
baum_element(knoten(El,L,R),El) ←
        baum_element(L,El),
        baum_element(R,El).
```

Beginnen wir mit der Optimierung der Kontextargumente in dem Prädikat baum_element/2. Die Transformation nach Kontinuationsprolog ergibt

```
[baum_element(nil,_El)] ↞
        [].
[baum_element(knoten(El,L,R),El)] ↞
        [baum_element(L,El),
        baum_element(R,El)].
```

Die Kontinuation enthält die redundante Variable El. Durch Einführen der Gleichung [baum_element(B,El)] $\doteq$ [baum_element(B,El),rest(El)] wird die Redundanz explizit gemacht, indem die Variable El verdoppelt wird. Es handelt sich um eine Transformation, die ein einzelnes Argument als Kontinuationselement für Optimierungen verfügbar macht. Durch Anwendung der Regeln R2 und R3 wird die Gleichung in das Programm compiliert und wir erhalten

```
[baum_element(nil,El),rest(El)] ↞
        [].
[baum_element(knoten(El,L,R),El),rest(El)] ↞
        [baum_element(L,El),
        rest(El),
        baum_element(R,El),
        rest(El)].
```

Das Kürzen der Kontinuation ist hier noch nicht möglich, da die Variable nicht ausschließlich in der zu kürzenden Kontinuation vorkommt. Es müssen also die Vorkommnisse der Variable El so umbenannt werden, daß Variablen der zu kürzenden

Kontinuation sonst nirgends in der Klause auftreten. Dies kann durch Anwendung der Regel R5, also Argumentgeneralisation erreicht werden. Wir erhalten also

```
[baum_element(nil,El),rest(_El)] ↞
        [].
[baum_element(knoten(El,L,R),El),~~rest(El2)~~] ↞
        [baum_element(L,El),
        rest(El1),
        baum_element(R,El1),
        ~~rest(El2)~~].
```

Das Prädikat entspricht nun der Form nach einem Programm von Listendifferenzen, wobei jeweils das letzte Argument von baum_element/2 und rest/1 ein Paar von Variablen darstellen, wie sie auch bei der Darstellung von Differenzen zu finden sind. Das nächste Beispiel zeigt explizit die Optimierung einer Differenz. Hier sei nur die strukturelle Gleichheit aufgezeigt, die es ermöglicht im weiteren wie bei der Optimierung von Differenzpaaren vorzugehen und die auch dem auf Seite 43 erläuterten Beispiel entspricht. Es bedarf aber noch der Anwendung einer Transformationsregel, um ein Programm zu erhalten, das direkt nach Low-Level Prolog übersetzt werden kann. Die Variable El2 tritt nur mehr in der zu kürzenden Kontinuation rest(El2) auf und wir können die Kontinuation rest(El2) kürzen. Wir erhalten also

```
[baum_element(nil,El),rest(_El)] ↞
        [].
[baum_element(knoten(El,L,R),El)] ↞
        [baum_element(L,El1),
        rest(El1),
        baum_element(R,El2),
```

Das Programm enthält somit kein Kontextargument mehr. Der redundante Ausdruck rest(El2) konnte gefunden und beseitigt werden. Nach der Übersetzung nach Low-Level Prolog zeigt das Programm die gewünschte Effizienzsteigerung. Die im vorigen Schritt erwähnte strukturelle Gleichheit mit Paaren von Differenzvariablen sei nun an einem weiteren Beispiel erläutert.

4.4.3 Optimierung von Differenzen

Differenzen kommen in mehrfacher Ausprägung in Prologprogrammen vor. Sie werden immer dann verwendet, wenn eine Folge von Zuständen in irgendeiner Form repräsentiert werden muß. Das Prädikat baum_summe/3 berechnet die Summe der Knoten eines Baumes, wobei jeder Knoten den Wert W enthält. Die Variablenpaare (N0,N),(N1,N2) und (N2,N) repräsentieren jeweils einen Zustandsübergang in der

Summierung von Knoten, wobei jede errechnete Summe für einen Teil des Baumes als ein Zustand (der Summe) angesehen werden kann.

```
baum_summe(nil,N,N).
baum_summe(knoten(W,Bl,Br),N0,N) ←
        N1 is N0+W,
        baum_summe(Bl,N1,N2),
        baum_summe(Br,N2,N).
```

Durch Übersetzung nach Kontinuationsprolog erhalten wir

```
[baum_summe(nil,N,N)].
[baum_summe(knoten(W,Bl,Br),N0,N)] ↞
        [N1 is N0+W,
        baum_summe(Bl,N1,N2),
        baum_summe(Br,N2,N)].
```

Die Transformation beginnt mit dem Einführen der Gleichung [baum_summe(B,N0,N)] $\doteq$ [baum_summe(B,N0),rest(N))] . Mit Hilfe dieser Gleichung wird das Differenzpaar in zwei Terme aufgespalten. Zwei neue Kontinuationselemente, baum_summe(B,N0) und rest(N) werden eingeführt. Durch Anwendung der Gleichung und der Regeln R2 und R3 wird das Prädikat baum_summe/3 in baum_summe/2 transformiert und wir erhalten

[baum_summe(nil,N),rest(N)] ↞
        [].
[baum_summe(knoten(W,Bl,Br),N0),~~rest(N)~~] ↞
        [N1 is N0+W,
        baum_summe(Bl,N1),
        rest(N2),
        baum_summe(Br,N2),
        ~~rest(N)~~].

Nun können die Kontinuationen durch Anwendung von R4 gekürzt werden und das Prädikat besteht somit aus folgenden Klausen

```
[baum_summe(nil,N),rest(N)] ↞
        [].
[baum_summe(knoten(W,Bl,Br),N0)] ↞
        [N1 is N0+W,
        baum_summe(Bl,N1),
        rest(N2),
        baum_summe(Br,N2)].
```

Die Anzahl der Argumente konnte somit um eins vermindert werden. Weiters konnte eine permanente Variable beseitigt werden, die somit nicht mehr am Stack gespeichert werden muß. Dieses Programm kann nun direkt nach Low-Level Prolog übersetzt werden oder nach Einführen eines Hilfsprädikates nach binärem Prolog.

4.5 Übersetzung von Kontinuationsprolog

Kontinuationsprolog kann nach verschiedenen abstrakten Maschinen übersetzt werden. Hier wird die Übersetzung nach binärem Prolog und Low-Level Prolog vorgestellt.

Die Übersetzung von Kontinuationsprolog nach binärem Prolog ist in jeder Phase des Transformationsprozesses möglich. Das in Kapitel 4 beschriebene Transformationssystem wurde erstmals in binärem Prolog formuliert [Neu92b]. Erst später wurde die Notation von Kontinuationsprolog eingeführt [Neu95b], die eine allgemeinere Darstellung von Prologprogrammen erlaubt.

Im Gegensatz zur Übersetzung nach binärem Prolog kann nur eine Untermenge von Kontinuationsprolog nach Low-Level Prolog übersetzt werden. Diese Untermenge ist dadurch charakterisiert, daß die WAM Stacks benutzt, um die Konjunktionen Prologs zu implementieren statt des allgemeineren Ansatzes von Kontinuationen.

4.5.1 Binäres Prolog

Binäres Prolog entspricht dem Continuation Passing Style (CPS) [Wan80] in funktionalen Sprachen. Die BinWAM [Tar91], eine vereinfachte reduzierte Form der WAM [War77], ist für die Ausführung von binären Prologprogrammen optimiert. Wegen der Existenz des Metacalls macht man in binärem Prolog keinen Unterschied zwischen Prädikaten- und Funktionssymbolen.

Binäres Prolog wird am schnellsten auf der dafür entwickelten Binary WAM (BinWAM) ausgeführt. In Hinsicht auf Aufrufkonventionen unterscheidet sich die BinWAM von der WAM, da jede Regel in binärem Prolog nur genau ein Ziel im Rumpf aufweist. Daher sind auch keine speziellen Mechanismen notwendig, um die

Rückkehr aus einer Prozedur entsprechend zu handhaben, sondern es kann immer sofort die nächste Regel aufgerufen werden. D.h, daß während des gesamten Programmablaufes alle temporären Werte in Registern gehalten werden können. Es können daher die gleichen Ergebnisse erzielt werden, wie sie in prodezuralen Sprachen nur mit Hilfe von Algorithmen zur optimalen Registerallokation möglich sind. Ein Nachteil ist allerdings, daß Datenstrukturen wie etwa Arrays von der BinWAM nicht effizient bearbeitet werden können.

In binärem Prolog enthält jede Klause nur genau ein Ziel im Rumpf. Eine Klause in binärem Prolog hat somit die Form

$$A \leftarrow B.$$

A ist der Kopf der Klause, B genau ein Ziel. Kontinuationsprolog hat aber beliebig viele Elemente sowohl im Kopf als auch im Rumpf einer Klause. Die Übersetzung von Kontinuationsprolog nach binärem Prolog ist Gegenstand des folgenden Abschnitts.

4.5.2 Übersetzung nach binärem Prolog

Bevor ein Kontinuationsprologprogramm nach binärem Prolog übersetzt werden kann, sind zwei Iterationen für Kontinuationstransformationen notwendig. Da alle vorbereitenden Transformationen in Kontinuationsprolog durchgeführt werden können, wird hier vorerst der Vorgang nur für Kontinuationsprolog erläutert. In einem Beispiel, beginnend auf Seite 53, werden auch die entsprechenden Gleichungen und Transformationsregeln in binärem Prolog angegeben. Im Folgenden wird die Definition eines Hilfsprädikates für das Lesen von Kontinuationen und das Übersetzen der Folge von Zielen im Rumpf einer Klause in die entsprechende Darstellung in binärem Prolog beschrieben. Um das benötigte Hilfsprädikat und dessen Verwendung im Programm zu definieren, sind zwei Iterationen notwendig.

Erste Iteration. Für alle Kontinuationsfunktoren $rest_i/n$ führen wir eine Gleichung [demo(X1,…,Xn),$rest_i$(X1,…,Xn)] $\doteqdot$ [] durch Anwendung von Regel R1 in das Programm ein. Mittels der Regeln R2 und R3 werden alle Gleichungen in das Programm compiliert bzw. hinzugefügt. Unter anderem wird das Hilfsprädikat demo/n eingeführt und der Aufruf von $rest_i/n$ durch den Aufruf des Hilfsprädikates demo/n ersetzt. Durch Argumentgeneralisation und Kürzen der Kontinuationselemente $rest_i/n$, also durch Anwendung der Regeln R5 und R4, werden alle Vorkommnisse dieser Kontinuationselemente bis auf das einmalige Vorkommen im Kopf des Hilfsprädikats demo/n beseitigt. Das Lesen von Kontinuationselementen im Kopf von Klausen wird nach diesem Iterationsschritt von einem Hilfsprädikat durchgeführt.

Zweite Iteration. In dieser Iteration werden unter anderem existentielle Variablen entfernt. Sei rest/m ein Kontinuationsfunktor und p/n ein beliebiger Prädikatsfunktor. Durch Anwendung von Regel R1 führen wir die Gleichungen [rest(X1,...,Xn),p(Y1,...,Yn)] $\doteqdot$ [rest_p(Z1,...,Zo)] ein, wobei $\{Z_1 \ldots Z_o\}$ eine Teilmenge der Vereinigung der Mengen $\{X_1 \ldots X_m\}$ und $\{Y_1 \ldots Y_n\}$ ist. Durch Anwendung von Regel R2 wird jeweils eine neue Klause demo/n für jedes Prädikat p/n erzeugt.

Anschließend wird das Kontinuationsprologprogramm nach binärem Prolog übersetzt, wobei ein neues Argument allen Prädikatsfunktoren hinzugefügt werden muß, um die Kontinuation zu repräsentieren. In dieser Kontinuation wird die Reihenfolge der einzelnen Ziele aufrechterhalten, indem jeder UND-Verknüpfung im Rumpf einer Klause eine Schachtelungstiefe der Kontinuation in binärem Prolog entspricht. Durch diese Schachtelung wird erreicht, daß nur genau ein Ziel im Rumpf jeder Klause vorkommt. Wir geben hier vorerst eine Definition für die Übersetzung eines Prologprogramms nach binärem Prolog von P. Tarau an [Tar91].

Definition. Sei P ein Programm und $Cont$ eine neue Variable. Seien T und $E = p(T_0 \ldots T_n)$ zwei Ausdrücke. Wir schreiben den Ausdruck $p(T_0 \ldots T_n, T)$ als $\Phi(E, T)$. Ausgehend von der Klause

$$A \leftarrow B_0 \ldots B_n.$$

erhalten wir die binäre Metaklause

$$\Phi(A, Cont) \leftarrow \Phi(B_1, \Phi(B_2, \ldots, \Phi(B_n, Cont) \ldots)).$$

Gemäß dieser Definition wird z.B. die Prologklause p(X) ←q(X),r(X) in p(X,Cont) ←q(X,r(X,Cont)) transformiert. Ebenso wird bei Kontinuationsprologklausen mit mehreren Zielen im Rumpf vorgegangen. Wir benötigen also noch eine Transformation, um Kontinuationen mit mehreren Elementen im Kopf nach binärem Prolog übersetzen zu können.

Für Klausen mit einem Kontinuationsfunktor im Kopf einer Klause ist es günstig, das Kontinuationselement als erstes Argument in der binären Klause zu verwenden. Dadurch wird eine spezielle Optimierung auf BinWAM-Ebene, nämlich das Indizieren des ersten Argumentes, ausgenutzt. Weiters ist zu beachten, daß das erste Element im Kopf einer Kontinuationsprologklause immer ein Prädikatsfunktor ist.

Beispiel. Gegeben sei die Klause

[p(X),rest(X)] ↞[q(X),r(X)].

Im Kopf tritt also der Kontinuationsfunktor rest/1 auf und im Rumpf haben wir eine Konjunktion von zwei Zielen. Durch Übersetzung nach binärem Prolog erhalten wir

```
p(rest(X,Cont),X) ←q(X,r(X,Cont)).
```

Das erste Element im Kopf war der Prädikatsfunktor p/1. Dieser wurde um ein Argument erweitert, das nun die Kontinuation enthält. Wie oben erwähnt, ist es günstig, das Kontinuationsargument an erster Stelle einzuführen, um das Indizieren des ersten Argumentes auf BinWAM-Ebene auszunützen. Da rest/1 ein Kontinuationsfunktor ist, wird er im ersten Argument, eben dem Kontinuationsargument, verwendet und, wie jeder andere Funktor auch, um ein Argument erweitert, um die Kontinuation aufzunehmen. Im Rumpf sehen wir die Schachtelung der Konjunktion der Ziele, welche der obigen Definition entspricht.

Wir geben für alle Schritte auch die entsprechende Darstellung in binärem Prolog an. Dadurch wird die Beziehung zwischen Kontinuationsprolog und binärem Prolog dargestellt und gezeigt, daß sämtliche Transformationsschritte auch in binärem Prolog durchführbar sind. Allerdings, und auch diesem Aspekt dient die folgende Darstellung, ist die Schreibweise in binärem Prolog weit unübersichtlicher und auch schwieriger zu implementieren. Die Hauptvorteile von Kontinuationsprolog sind: Klarheit der Darstellung und einfache Implementierung.

Das Prädikat ausdruck/3 beschreibt das Einebnen eines binären Baumes in eine Liste von Prefixausdrücken. Ein binärer Baum wird eingeebnet, indem eine Folge von Elementen gefunden wird, in der ein Operator, der für einen Knoten des Baumes steht, angibt, daß das erste folgende Element den linken und das zweite folgende Element den rechten Unterbaum darstellt. Beide Elemente können wiederum Operatoren enthalten. Das Prädikat sieht folgendermaßen aus:

```
ausdruck(b(W),[b(W)|Xs],Xs).
ausdruck(knoten(Bl,Br),[op|Xs0],Xs) ←
        ausdruck(Bl,Xs0,Xs1),
        ausdruck(Br,Xs1,Xs).
```

Beispielsweise entspricht dem Baum knoten(knoten(b(3),b(2)),b(1)) in Prefixschreibweise die Liste [op,op,b(3),b(2),b(1)].

Die Übersetzung nach Kontinuationsprolog ergibt

```
[ausdruck(b(W),[b(W)|Xs],Xs)] ↞
        [].
[ausdruck(knoten(Bl,Br),[op|Xs0],Xs)] ↞
        [ausdruck(Bl,Xs0,Xs1),
        ausdruck(Br,Xs1,Xs)].
```

Die Klause enthält im Rumpf also die Konjunktion von zwei Zielen. Die entsprechende Darstellung in binärem Prolog ist nun um ein Argument für die Kontinuation erweitert und hat folgende Gestalt

> ausdruck(b(W),[b(W)|Xs],Xs,$\underline{\text{Cont}}$) $\leftarrow$
> $\underline{\text{Cont}}$.
> ausdruck(knoten(Bl,Br),[op|Xs0],Xs,$\underline{\text{Cont}}$) $\leftarrow$
>   ausdruck(Bl,Xs0,Xs1,$\underline{\text{ausdruck(Br,Xs1,Xs,}\underline{\text{Cont}}\text{)}}$).

Die Kontinuationen sind unterstrichen dargestellt. Die zweite Klause enthält nun nur mehr ein Ziel im Rumpf. Die Kontinuation enthält jeweils den Teil einer Klause, der noch auszuführen ist. In diesem Fall ist dies ausdruck(Br,Xs1,Xs,Cont). Es muß also noch der für Kontinuationsprolog spezielle Fall behandelt werden, daß im Kopf mehrere Elemente vorkommen können. Beginnen wir aber zuerst mit der Anwendung einiger Transformationsregeln bis dieser Fall eintritt.

Die durch die Variablen Xs0,Xs1 und Xs beschriebenen Listendifferenzen werden im Folgenden gemäß der in 4.4.3 beschriebenen Strategie optimiert.

Wird dieses Programm auf Ebene von Kontinuationsprolog optimiert, so fahren wir mit Einführen der Gleichung [ausdruck(B,Xs0,Xs] $\underset{\cdot}{=}$ [ausdruck1(B,Xs0),rest(Xs)] fort. Die entsprechende Gleichung zur Transformation eines Programms in binärem Prolog lautet [ausdruck(B,Xs0,Xs,Cont)] $\underset{\cdot}{=}$ [ausdruck1(B,Xs0,rest(Xs,Cont))] . Der wesentliche Unterschied liegt in der vorhandenen expliziten Kontinuation *Cont* und darin, daß rest(Xs,Cont) selbst Kontinuation ist und daher hier als letztes Element auftritt.

Es ist nun möglich sämtliche Optimierungen in binärem Prolog durchzuführen, da zu allen Transformationsregeln und -gleichungen die entsprechende auch in binärem Prolog angegeben werden kann. Daraus folgt aber auch, daß alle Transformationen in Kontinuationsprolog durchgeführt werden können und erst am Ende nach binärem Prolog übersetzt werden muß. Dabei ist zu beachten, daß die Übersetzung nach Low-Level Prolog (siehe 4.5.3 auf Seite 59) an anderer Stelle der Folge von Transformationsschritten durchgeführt wird, als die Übersetzung nach binärem Prolog. Der hier beschriebene Transformationsprozeß ist für die Übersetzung nach binärem Prolog entsprechend erweitert. Der Hauptunterschied zur Übersetzung nach Low-Level Prolog liegt im Einführen eines Hilfsprädikates.

Fahren wir also fort mit der Transformation. Die Interface-Regel

> [ausdruck(B,Xs0,Xs)] $\twoheadleftarrow$
>   [ausdruck1(B,Xs0,rest(Xs,Cont))].

ist nötig, um das neue Prädikat ausdruck1/3 in das Programm einbinden zu können. Compiliert man obige Gleichung in das Programm, d.h wendet man die Transformationsregeln R2 und R3 an, so erhält man

```
[ausdruck1(b(W),[b(W)|Xs]),rest(Xs)] ↞
        [].
[ausdruck1(knoten(Bl,Br),[op|Xs0],rest(Xs))] ↞
        [ausdruck1(Bl,Xs0)
        rest(Xs1),
        ausdruck1(Br,Xs1),
        rest(Xs)].
```

Dieses Programm zeigt nun klar die redundanten Ausdrücke. Der Term rest(Xs) wird vom Kopf gelesen, um unverändert im Rumpf wieder geschrieben zu werden. Wir können diesen Ausdruck durch Anwendung von Regel R4 kürzen, d.h., daß der Ausdruck im Kontinuationsprologprogramm vollständig beseitigt wird.

Scheinbar verschieden von dieser Art der Elimination redundanter Ausdrücke ist die Vorgangsweise in binären Prologprogrammen. Hier folgt nun das dem obigen Kontinuationsprologprogramm entsprechende binäre Prologprogramm:

```
ausdruck(B,Xs0,Xs,Cont) ←
        ausdruck1(B,Xs0,rest(Xs,Cont)).

ausdruck1(b(W),[b(W)|Xs],rest(Xs,Cont)) ←
        Cont.
ausdruck1(knoten(Bl,Br),[op|Xs0],rest(Xs,Cont)) ←
        ausdruck1(Bl,Xs0,rest(Xs1,ausdruck1(Br,Xs1,rest(Xs,Cont)))).
```

Die Kontinuation der zweiten Regel enthält im Rumpf noch eine weiter Subkontinuation. Dem Kürzen von Kontinuationselementen in Kontinuationsprolog entspricht in binärem Prolog das Ersetzen von Kontinuationsausdrücken durch Variablen. Wir erhalten

```
ausdruck(B,Xs0,Xs,Cont) ←
        ausdruck1(B,Xs0,rest(Xs,Cont)).

ausdruck1(b(W),[b(W)|Xs],rest(Xs,Cont)) ←
        Cont.
ausdruck1(knoten(Bl,Br),[op|Xs0],RestXsCont) ←
        ausdruck1(Bl,Xs0,rest(Xs1,ausdruck1(Br,Xs1,RestXsCont))).
```

Die erste Regel ist wieder die Interface-Regel. Der Ausdruck rest(Xs,Cont) wurde durch die Variable RestXsCont ersetzt. Das Problem, das sich nun noch stellt, ist das Vorkommen von rest(Xs,Cont) im Kopf der ersten Klause von ausdruck1/3. Um auch diesen Ausdruck entfernen zu können, ist es notwendig ein Hilfsprädikat einzuführen, das die Kontinuation liest. Dies ist auch genau der Punkt, an dem die Übersetzung nach Low-Level Prolog einen anderen Weg nimmt, da für die Übersetzung nach Low-Level Prolog kein Hilfsprädikat notwendig ist. Dennoch können auch diese Schritte in Kontinuationsprolog durchgeführt werden. Hier sei noch das dem obigen binären Prologprogramm entsprechende Kontinuationsprologprogramm angegeben, das direkt nach Low-Level Prolog übersetzt werden kann.

```
[ausdruck(B,Xs0,Xs)] ↞
        [ausdruck1(B,Xs0),rest(Xs)].

[ausdruck1(b(W),[b(W)|Xs]),rest(Xs)] ↞
        [].
[ausdruck1(knoten(Bl,Br),[op|Xs0])] ↞
        [ausdruck1(Bl,Xs0),
        rest(Xs1)
        ausdruck1(Br,Xs1)].
```

Für die Übersetzung nach binärem Prolog wird ein Hilfsprädikates definiert. Wir geben zuerst das binäre Prologprogramm an:

```
ausdruck1(b(W),[b(W)|Xs],RestXsCont) ←
        demo(RestXsCont,Xs).
ausdruck1(knoten(Bl,Br),[op|Xs0],RestXsCont) ←
        ausdruck1(Bl,Xs0,rest(Xs1,ausdruck1(Br,Xs1,RestXsCont))).

demo(rest(Xs,Cont),Xs) ←
        Cont.
```

Die Transformation in binärem Prolog besteht darin, daß rest/2 durch demo/2 ersetzt und das neue Prädikat demo/2 eingeführt wird, sodaß es die Kontinuation lesen kann.

Die in Kontinuationsprolog notwendige Gleichung ist

$$[\mathrm{demo(Xs)},\mathrm{rest(Xs)}] \doteq []$$

Die Gleichung ist notwendig, um den Aufruf des Hilfsprädikates in der ersten Klause einzuführen und um das Hilfsprädikat demo/1 einzuführen. Wie in 4.3.2 auf Seite 38 bereits erwähnt, wird mittels einer Gleichung $u \cdot v = []$ durch Anwendung der Transformationsregel R2 eine neue Klause $[u \cdot v] = []$ erzeugt; d.h., mittels dieser Gleichung wird das Hilfsprädikat eingeführt und wir erhalten somit in Kontinuationsprolog

$$
\begin{array}{l}
\text{[demo(Xs),rest(Xs)]} \twoheadleftarrow \\
\qquad \text{[].}
\end{array}
$$

An diesem Fall sieht man auch, wie zwei Elemente im Kopf einer Kontinuationsprologklause nach binärem Prolog übersetzt werden können. Der Funktor des ersten Elements wird Funktor des Prädikates, ist also der Prädikatsname. Die erste Argumentposition wird nun nicht von der Variable eingenommen, sondern vom Kontinuationselement, um die auf BinWAM-Ebene vorhandene Optimierung der Indizierung des ersten Argumentes auszunützen. Dann erst folgt das Argument des Prädikates, welches in diesem Fall die Variable Xs ist.

Nun folgt der letzte Schritt, welchem alle bisherigen Schritte als Vorbereitung gedient haben: Das Beseitigen der existentiellen Variable Xs1. Die Gleichung lautet:

$$\text{[rest(Xs1),ausdruck1(Br,Xs1)]} \underset{\cdot}{=} \text{[rest\_ausdruck1(Br)]}$$

bzw. in binärem Prolog

$$\text{rest(Xs1,ausdruck1(Br,Xs1,RestXsCont))} \underset{\cdot}{=} \text{rest\_ausdruck1(Br,RestXsCont)}$$

Das Kontinuationselement rest/1 tritt nun nicht mehr im Prädikat ausdruck1/2 auf, sondern ausschließlich im Hilfsprädikat demo/2. ausdruck1/2 hat in binärem Prolog schlußendlich folgendes Aussehen:

$$
\begin{array}{l}
\text{ausdruck(B,Xs0,Xs,}\underline{\text{Cont}}\text{)} \leftarrow \\
\qquad \text{ausdruck1(B,Xs0,}\underline{\text{rest(Xs,Cont)}}\text{).} \\
\\
\text{ausdruck1(b(W),[b(W)|Xs],}\underline{\text{RestXsCont}}\text{)} \leftarrow \\
\qquad \text{demo(}\underline{\text{RestXsCont}}\text{,Xs).} \\
\text{ausdruck1(knoten(Bl,Br),[op|Xs0],}\underline{\text{RestXsCont}}\text{)} \leftarrow \\
\qquad \text{ausdruck1(Bl,Xs0,}\underline{\text{rest\_ausdruck1(Br,}\underline{\text{RestXsCont}}\text{)}}\text{).}
\end{array}
$$

In Kontinuationsprolog erhalten wir durch Anwendung obiger Gleichung auf [demo(Xs),rest(Xs)] $\twoheadleftarrow$[] eine weitere Klause. Die Transformationsregel R2 kommt zur Anwendung und gemäß Ersetzungsregel 2 für überlappende Elemente von Kontinuationsgleichungen wird dem Hilfsprädikat demo/2 eine Klause mit dem Rumpf [ausdruck1(Br,Xs)] hinzugefügt. Das Hilfsprädikat hat in Kontinuationsprolog folgende Klausen:

```
[demo(Xs),rest(Xs)] ↞
        [].
[demo(Xs),rest_ausdruck1(Br)] ↞
        [ausdruck1(Br,Xs)].
```

Bei der Übersetzung nach binärem Prolog wird nun die Kontinuation wieder im ersten Argument angeordnet. Daher wird der Term rest_ausdruck1 in das erste Argument gestellt.

```
demo(rest(Xs,Cont),Xs) ←
        Cont.
demo(rest_ausdruck1(Br,Cont),Xs) ←
        ausdruck1(Br,Xs,Cont).
```

Für die Anfrage

```
← ausdruck(knoten(knoten(t(3),t(2)),t(1)),B,[],true).
```

erhalten wir B=[op,op,t(3),t(2),t(1)], was dem Ergebnis des ursprünglichen Prologprogramms entspricht.

Folgende Optimierungen konnten somit erreicht werden:

- Die aktuelle Liste wird nun nicht mehr durch verschiedene Variablen repräsentiert und durch diese weitergereicht, sondern verbleibt im selben Argumentregister. Das Lesen der Liste konnte dem Verhalten eines prozeduralen Programms angenähert werden.
- Die Kontinuation konnte gekürzt werden, da alle existentiellen Variablen X1 und Xs entfernt werden konnten. Daher werden auch keine Trail-Überprüfungen bzgl. der Bindung dieser Variablen mehr durchgeführt (siehe 5.1.6 auf Seite 72).
- Ein Argumentregister konnte eingespart werden. Daher wird eine Interface-Regel benötigt, um das optimierte Prädikat in das Programm einzubinden.
- Auf einer BinWAM wird das optimiert Programm um 70% schneller ausgeführt [Neu95].

4.5.3 Low-Level Prolog

Low-Level Prolog ist eine Zwischensprache in der Übersetzung nach WAM-Code. Zwei built-in Prädikate werden in das Prologprogramm eingeführt, sodaß man durch die Übersetzung von Low-Level Prolog nach der abstrakten Maschine WAM-Code erhält, dessen Ausführung eine effizientere Allokation von Registern zeigt.

Die Untermenge von Kontinuationsprolog, die für die Übersetzung geeignet ist, wird durch folgende Bedingungen beschrieben:

1. Alle Funktoren von Kontinuationselementen müssen im Kopf einer Klause entweder an erster oder an zweiter Stelle auftreten.

2. Funktoren, die an erster Stelle auftreten sind Prädikatsfunktoren, jene die an zweiter Stelle auftreten sind Kontinuationsfunktoren.

3. Wenn eine Klause einen Kontinuationsfunktor im Kopf aufweist, muß der Rumpf der Klause leer sein.

4. Der Kontinuationsfunktor muß für alle Klausen eines Prädikates der gleiche sein.

Kontinuationsprologprogramme werden also in Low-Level Prolog, erweitert um die built-in Prädikate 'SOURCE' und 'SINK', übersetzt. Die beiden built-in Prädikate werden in WAM Befehle übersetzt, sodaß keine Erweiterung der WAM notwendig ist. Eine experimentelle Version dieser built-in Prädikate wurde von U. Neumerkel erstmals in [Neu96] vorgestellt. Beide Prädikate können eine beliebige Anzahl an Argumenten aufweisen.

SINK/n : sichert zu, daß sich alle Argumente in den entsprechenden Registern zum Zeitpunkt des Aufrufes befinden. Die Nummer des Argumentes entspricht dabei der Nummer des Argumentregisters. Es muß nur als letztes Ziel einer Klause auftreten. Der Befehl ist ähnlich einem gewöhnlichen Ziel (bzgl. des generierten WAM-Codes)

SOURCE/n : sichert zu, daß alle Argumente die entsprechenden Register verwenden. In einer Klause tritt es nur direkt nach einem gewöhnlichen Ziel auf, d.h. im WAM-Code unmittelbar nach einer call Instruktion. Die Implementation des Befehles umfaßt eine komplexe Allokation von Registern im Rumpf einer Klause.

Für einige Optimierungen muß nur zugesichert werden, daß ein Teil der Argumente in den entsprechenden Registern steht, d.h. es ist für manche Argumentpositionen nicht entscheidend, welcher Wert sich im entsprechenden Register befindet. Um dies auszudrücken, ist es möglich Argumentpositionen als definiert oder undefiniert zu kennzeichnen. '+(X)' bedeutet, daß das Argument den Wert X an der Position, an der der Term auftritt, enthält. Ein '–' zeigt an, daß diese Argumentposition undefiniert ist.

4.5.4 Übersetzung nach Low-Level Prolog

Die Übersetzung von Kontinuationsprolog nach Low-Level Prolog muß nur die richtige Verwendung der built-in Prädikate gewährleisten. Allerdings sind zusätzliche Überlegungen bzgl. der Arität von 'SINK' und 'SOURCE' notwendig, um eine optimale Registerallokation zu gewährleisten. Die Folgenden drei Regeln beschreiben die Übersetzung, legen aber nur fest, wo diese beiden Prädikate verwendet werden, nicht aber deren Arität und die Verwendung der Variablen in diesen.

1. Prädikatfunktoren werden direkt in Kopf und Rumpf abgebildet. Das bedeutet auch, daß die Arität der Prädikate unverändert bleibt.

2. Kontinuationsfunktoren im Kopf einer Klause werden auf 'SINK'/n abgebildet.

3. Kontinuationsfunktoren im Rumpf einer Klause werden auf 'SOURCE'/n abgebildet.

Bei dieser Transformation ist eine Optimierung der dritten Low-Level-Transformationsregel möglich: Tritt ein Kontinuationsfunktor der Form rest/1 als letztes Element auf, so kann auf das built-in Prädikat 'SINK'/n abgebildet werden. Da 'SINK'/n im Vergleich zu 'SOURCE'/n der schnellere Befehl ist, steigert diese Optimierung die Effizienz des Programms.

Die oben angesprochenen nötigen zusätzlichen Überlegungen werden nun an Hand eines Beispiels durchgeführt. Im Folgenden wird das in 4.5.2 optimierte Programm zur Einebnung eines binären Baumes nach Low-Level Prolog übersetzt. Das optimierte Programm wird hier nochmals angegeben:

```
[ausdruck(B,Xs0,Xs)] ↞
        [ausdruck1(B,Xs0),rest(Xs)].

[ausdruck1(b(W),[b(W)|Xs]),rest(Xs)] ↞
        [].
[ausdruck1(knoten(Bl,Br),[op|Xs0])] ↞
        [ausdruck1(Bl,Xs0),
        rest(Xs1)
        ausdruck1(Br,Xs1)].
```

An dieser Stelle des Optimierungsprozesses, ist in binärem Prolog nun das Einführen eines Hilfsprädikates zur Darstellung der Kontinuation notwendig. Für die Übersetzung von Kontinuationsprolog nach Low-Level Prolog ist keine weitere Transformation des Programms auf Ebene von Kontinuationsprolog notwendig. Es kann direkt nach Low-Level Prolog übersetzt werden. Wir erhalten also

```
ausdruck(B,Xs0,Xs) ←
        ausdruck1(B,Xs0),
        'SOURCE'(-,+Xs).

ausdruck1(b(W),[b(W)|Xs]) ←
        'SINK'(-,+Xs).
ausdruck1(knoten(Bl,Br),[op|Xs0]) ←
        ausdruck1(Bl,Xs0),
        'SOURCE'(-,+Xs1),
        ausdruck1(Br,Xs1).
```

Das Programm wurde durch Anwendung der Regel 1 bis auf die Kontinuationsfunktoren kopiert. Die Verwendung der built-in Prädikate ist gemäß der Low-Level-Transformationsregeln 2 und 3 erfolgt. Die Arität von 'SOURCE' und 'SINK' und die Anordnung der Variablen in diesen bedarf weiterer Überlegungen.

Durch 'SINK' in der ersten Klause von ausdruck1/2 wird zugesichert, an welcher Argumentposition der Rückgabewert der Variablen Xs steht. In der zweiten Klause wird an der ersten Argumentposition im rekursiven Aufruf von ausdruck1/2 der linke bzw. rechte Baum repräsentiert. Um zu gewährleisten, daß die Liste des eingeebneten Baumes während der gesamten Ausführung des Prädikates im selben Argumentregister steht, muß die zweite Argumentposition für die Variable X verwendet werden. Durch 'SOURCE'/2 wird zugesichert, daß der Rückgabewert des zweiten Argumentes in der Variable Xs1 steht und somit unverändert im zweiten Aufruf von ausdruck1/2 verwendet werden kann. Was sich nach dem ersten Aufruf an der ersten Argumentposition befindet, ist unbedeutend, da beim zweiten Aufruf von ausdruck1/2 an erster Stelle Br steht. Dies wird durch Verwendung von '–' zum Ausdruck gebracht.

Damit ist der Transformationsprozeß abgeschlossen.

Zusammenfassung

In diesem Kapitel haben wir Transformationen kennengelernt, die angewendet werden, um die interprozedurale Registerallokation von Prolog der von imperativen Sprachen anzunähern. Sämtliche Optimierungen wurden in Kontinuationsprolog durchgeführt. Durch die Übersetzung des optimierten Programms nach einer abstrakten Maschine erreichten wir ein verbessertes Laufzeitverhalten.

Die Transformationen wurden durch fünf Regeln beschrieben. Die Anwendung dieser veränderte weder die operationale Semantik des ursprünglichen Programms noch das Verhalten der syntaktischen Unifikation. Drei wesentliche Strategien zur Anwendung der Transformationsregeln wurden erläutert: Einführen von Kontextargumente, Optimierung von Kontextargumenten und Optimierung von Differenzvariablen. Diese sind gemäß der Reihenfolge, in der sie hier angegeben sind, auch jeweils ineinander überführbar, wie dies in Abbildung 4.2 dargestellt wurde. Durch diese Optimierungen wurde eine Reduktion der Anzahl der Argumente erreicht. Die verminderte Zahl an permanenten Variablen hatte eine Verkleinerung der Wahlpunkte zur Folge, sodaß weniger Daten auf und vom Stack bewegt werden mußten.

Weiters untersuchten wir die Übersetzung des optimierten Programms nach binärem Prolog und Low-Level Prolog. Ein binäres Prologprogramm bestand aus Klausen mit einem Element im Kopf und genau einem Element im Rumpf. Prädikatsfunktoren mußten um ein Argument erweitert werden, um die Kontinuation darzustellen. Um ein optimiertes Kontinuationsprologprogramm nach binärem Prolog übersetzen zu können, war es notwendig ein Hilfsprädikat einzuführen, das die Kontinuationselemente lesen konnte. Die Ausführung auf einer BinWAM brachte dann die gewünschte Effizienzsteigerung. Low-Level Prolog war ein um die built-in Prädikate 'SOURCE'/n und 'SINK'/n erweitertes Prologprogramm. Durch die Verwendung dieser Prädikate konnten wir die interprozedurale Registerallokation der WAM dahingehend beeinflussen, daß Werte länger und über Prädikatsgrenzen hinweg im selben Register gehalten werden konnten. Low-Level Prologprogramme waren für die Übersetzung nach der WAM geeignet. Es war keine Erweiterung oder Veränderung des Befehlssatzes der WAM erforderlich, da 'SOURCE'/n und 'SINK' in Befehle der WAM übersetzt werden konnten.

Das optimierte Programm wies eine verbesserte interprozedurale Registerallokation auf. Es wurden weniger Register zur Darstellung logischer Variablen verwendet, sodaß Programmkonstrukte wie Kontextargumente und Differenzen effizienter implementiert werden konnten. Die Registerallokation der WAM wurde der eines prozeduralen Programms stark angenähert.

Im folgenden Kaptitel werden wir die Begriffe Wahlpunkt, Environment und permanente Variable genauer erläutern. Es sind dies Elemente der WAM, deren Optimierung starken Einfluß auf die Effizienz eines Prologprogramms hat. Wir beschreiben den Aufbau und die Funktionsweise der WAM, um dann abschließend die Optimierungen, die durch Kontinuationstransformationen erreicht werden konnten, auch in Begriffen der WAM erklären zu können.

Kapitel 5

Warren Abstract Machine

Im Jahre 1983 präsentierte David H.D. Warren eine abstrakte Maschine, die eine Speicherarchitektur und einen Befehlssatz enthält, um Prolog effizient ausführen zu können [War83]. Dieses Design wurde unter dem Namen Warren Abstract Machine bekannt (WAM). Die WAM ist zwar nicht der einzige existierende Ansatz, Prolog zu implementieren, aber jener, der in den letzten Jahren am intensivsten untersucht wurde.

Es gibt zwei Ansätze für Prologimplementation: emulierter Code, d.h. ein Prologprogramm wird nach einer Zwischensprache übersetzt, oder compilierter Code, d.h. Prolog wird direkt nach Maschinencode übersetzt. Ersterer ist kompakter in der Darstellung aber nicht so effizient wie der zweite Ansatz. WAM-Code ist emulierter Code. Die Klarheit und Übersichtlichkeit dieser Darstellung erleichterte die Diskussion und Untersuchung von Optimierungen und muß mit als Grund dafür gesehen werden, daß die WAM ein Standard für die Implementierung von Prolog geworden ist.

Im Folgenden werden die Datenstrukturen der WAM vorgestellt. Der Zusammenhang zu den in Kapitel 4 erläuterten Kontinuationstransformationen besteht darin, daß jede Optimierung auf Prolog-Ebene auch in Zusammenhang mit den Datenstrukturen der WAM gesehen werden muß.

5.1 Aufbau

Die WAM besitzt mehrere Stacks, von denen ein Teil auch in Ausführungsmodellen von imperativen Sprachen zu finden ist, wie etwa ein Stack zur Verwaltung von Rücksprungadressen bei Aufrufen von Prozeduren oder die Verwaltung lokaler Variablen. Weiters existieren lokale und globale Register. Dieses Design ist noch um Befehle und Datenstrukturen erweitert, um Unifikation und Backtracking zu implementieren.

H	Ende des Heaps, bzw. Zeiger auf nächsten Befehl
A	Ende des lokalen Stack
TR	Ende des Trail
CP	Ende des Stacks bei Rückkehr aus einer Klause
S	Heap Adresse - Zeiger auf Struktur
B	Adresse, an der im Falle von Backtracking fortgesetzt wird
HB	Heap Adresse, an der im Falle von Backtracking fortgesetzt wird

Tabelle 5.1: Übersicht über globale Register der WAM

5.1.1 Register

Die WAM verfügt über zwei Arten von Registern. Lokale Register werden bei der Unifikation und bei der Behandlung von Prozeduren verwendet. Globale Register sind vorhanden, um die Stack-Architektur der WAM als auch einige Optimierungen effizient zu implementieren.

Lokale Register

Während der Unifikation ist es nötig, Teile von Termen zwischenzuspeichern. Aus diesem Grund existieren Register, um Heapdaten temporär zu speichern, während Terme am Heap erstellt bzw. unifziert werden. Lokale Register haben daher das Format von Heapzellen (siehe auch 5.2).

Globale Register

Ein Großteil der globalen Register wird benötigt, um die Stacks der WAM zu verwalten. Aus Effizienzgründen stehen auch einige Programmadressen in globalen Registern. So wird etwa im Register CP die Rücksprungadresse für den Aufruf einer Prozedur gespeichert. In Tabelle 5.1 ist eine Übersicht globaler Register der WAM angegeben. Register H enthält die Adresse des Heapendes. Das Register HB beinhaltet immer den Wert von H, der im jüngsten Wahlpunkt gespeichert ist. Die Register A und Br werden verwendet, um das Ende (bzw. die Größe) des lokalen Stacks und des Trail-Stacks zu verwalten. Weiters finden bei Backtracking die Register B und HB Verwendung. In imperativen Sprachen als Rücksprungadresse bezeichnet, gibt in der WAM das Register CP die Adresse an, an der nach (erfolgreichem) Aufruf einer Prozedur fortgesetzt wird. Das Register S wird während der Unifikation von zusammengesetzten Termen mit Argumenten verwendet, um auf das jeweilige zu unifizierende Argument zu verweisen. Alle Argumente können somit angesprochen werden, indem S um die Größe einer Heapzelle erhöht wird, da die einzelnen Argumente immer direkt aufeinanderfolgend am Heap gespeichert werden.

Laden von Argumentregistern
put_structure f/n, Ai set_variable Xn,Ai set_value Xn,Ai

Unifikation mit Registern
get_structure f/n,Ai get_variable Xn,Ai get_value Xn,Ai

Unifikation mit Argumenten von Strukturen
unify_variable Xn,Ai unify_value Xn,Ai

Tabelle 5.2: Grundlegende Befehle für Unifikation

5.1.2 Befehlssatz

Die Befehle der WAM können gemäß ihrer Verwendung in drei Gruppen unterteilt werden:

- Unifikation (get,put und unify Befehle)
- Prozeduren (call,return und jump Befehle)
- Backtracking (try, retry und trust Befehle)

Die Unifikation einer Variable mit einem Term, der zur Zeit des Compilierens bekannt ist, wird in Befehle aufgeteilt, sodaß der Funktor und die Argumente getrennt behandelt werden. Prädikate werden wie Prozeduren behandelt. Daher sind Sprungbefehle vorhanden, um das Aufrufen und die Rückkehr aus Prozeduren zu ermöglichen. Für Backtracking werden drei verschiedene Befehle verwendet, die Wahlpunkte am lokalen Stack erzeugen und beseitigen.

5.1.3 Unifikation

Die Unifikation subsummiert die Wertzuweisung und die Parameterübergabe imperativer Programmiersprachen. Durch Unifikation zweier Terme, werden die Variablen, die in diesen vorkommen, gebunden. Um zwei Terme zu unifizieren, muß auf dem Heap eine Datenrepräsentation erzeugt werden, um die Bindungen von Variablen darstellen zu können. Diese Termdarstellung wird nun erläutert.

00	STR	1
01	g/1	
02	REF	2
03	STR	4
04	f/1	
05	REF	2
06	STR	7
07	p/2	
08	STR	4
09	STR	1

Tabelle 5.3: Heapdarstellung von p(f(X),g(X))

Termdarstellung

Ein Term wird in eine Folge von WAM-Befehlen compiliert. Die Ausführung dieser erzeugt dann eine entsprechende Repräsentation des Terms am Heap. Aufgrund des baumartigen Aufbaus eines Terms werden Teile in Registern zwischengespeichert. Die Reihenfolge der dazu nötigen Schritte ist implizit in der Reihenfolge der WAM-Befehle festgelegt. Durch WAM-Befehle wird der Name und die Arität des Funktors einer Struktur festgehalten und, falls nötig, am Heap erzeugt. Da in einem Term eine Variable mehrfach auftreten kann, müssen zwei Fälle unterschieden werden: Das erstmalige Auftreten einer Variable hat die Festlegung ihrer Repräsentation in einem Register oder am Heap zur Folge, jedes weitere wiederholte Auftreten einer Variable bedarf ihrer vollständigen Unifikation. Eine Übersicht über grundlegende Befehle für Unifikation gibt Tabelle 5.2. Durch diese Befehle wird eine Repräsentation zweier Terme am Heap erzeugt. Für die Darstellung der Unifikation verwenden wir ein vereinfachtes Modell der WAM. Einer der Terme wird als Anfrage auf den Heap geschrieben. Dann wird der Programmcode ausgeführt und beide Terme unifiziert.

In Abbildung 5.3 ist die Heapdarstellung der Anfrage $\leftarrow$ p(f(X),g(X)) zu sehen. Die Darstellung des Terms beginnt bei Adresse 6. Dieses einfache Modell eines Heaps verfügt nur über drei Arten von Heapzellen: 'STR'-Zellen bezeichnen eine Struktur. Funktorzellen beschreiben den Namen eines Funktors und dessen Arität. 'REF'-Zellen repräsentieren die Variablen. Die in diesem Beispiel verwendete Variable X tritt zweimal im Term auf und ist deswegen durch zwei Zellen am Heap repräsentiert. Die Bindung der beiden Zellen ist durch den Verweis der Zelle 5 auf die Zelle 2 dargestellt. Da die Variable X an keinen Term gebunden ist, verweist die Zelle 2 auf sich selbst.

Der Code für die Anfrage $\leftarrow$ p(g(X),f(X)) ist in Tabelle 5.4 auf der nächsten Seite dargestellt. 'put'- und 'set'-Befehle schreiben die Datenstrukturen für f/1, g/1 und p/2 auf den Stack. Die Variable X tritt zweimal in der Anfrage auf. Durch den

Programmcode	Modus	H
put_structure g/1,X3	schreiben	2
set_variable X4	schreiben	3
put_structure f/1,X2	schreiben	5
set_value X4	schreiben	6
put_structure p/2,X1	schreiben	8
set_value X2	schreiben	9
set_value X3	schreiben	10

Tabelle 5.4: WAM-Code der Anfrage p(g(X),f(X))

Programmcode	Modus	S	H
get_structure p/2,X1	lesen	8	10
unify_variable X2	lesen	9	10
unify_variable X3	lesen	10	10
get_structure f/1,X2	lesen	5	10
unify_variable X4	lesen	6	10
get_structure a/0,X4	schreiben	6	12
get_structure g/1,X3	lesen	2	12
unify_variable X5	lesen	3	12
get_structure a/0,X5	lesen	12	12

Tabelle 5.5: WAM-Code des Terms p(f(a),g(a)).

Befehl set_variable X4 wird an Adresse 2 eine ungebundene, d.h auf sich selbst verweisende, Zelle reserviert und der Inhalt dieser Zelle in das Register X4 geladen. Gebunden werden diese beide Zellen durch den Befehl set_value X4, der den Inhalt des Registers X4 auf den Heap an Adresse 5 schreibt.

Nachdem Daten einer Anfrage auf den Heap geschrieben worden sind, wird das Programm ausgeführt. In Tabelle 5.5 ist der Programmcode für den Term p(f(a),g(a)) angegeben. Bei der Ausführung des Programms werden die Daten der Anfrage am Heap durch das Programm unifiziert. Die WAM ist im Lese-Modus, wenn Teile des Terms, die durch das Programm unifiziert werden, bereits am Heap stehen. Durch die Anfrage ←p(g(X),f(X)) wurden bereits Teile des Terms p/2 am Heap erzeugt. Ist es notwendig, weitere Daten am Heap zu erzeugen, so geht die WAM in den Schreib-Modus über. Dies ist bei Schreiben des Terms a/0 der Fall. Die Variable X wird an den Term a/0 gebunden. Während der Unifikation zweier Terme gibt das Register S an, an welcher Stelle des Heaps der jeweils nächste zu unifizierende Term steht (In der Tabelle ist immer der Wert *nach* Ausführung des Befehles angegeben). So verweist etwa das Register S nach Ausführung des Befehls put_structure p/2,X1 auf das erste Argument von p/2. Dieses steht an Adresse 8 am Heap. Das globale

00	STR	1
01	g/1	
02	REF	10
03	STR	4
04	f/1	
05	REF	2
06	STR	7
07	p/2	
08	STR	4
09	STR	1
10	STR	11
11	a/0	

Tabelle 5.6: Heapdarstellung von p(f(a),g(a))

Register H verweist immer auf das Ende des Heaps. Der Wert dieses Registers ändert sich, wenn Zellen am Heap erzeugt werden und sich daher das Ende des Heaps ändert. Zellen werden am Heap erzeugt, wenn die WAM in den Schreib-Modus übergeht. Daher wird durch das Schreiben der Struktur a/0 auf den Heap das Register H erhöht. Da für die Struktur a/0 zwei Zellen erzeugt werden, wird das Register H von 10 auf 12 erhöht.

Nach Ausführung des Programms sind beide Terme unifiziert. Der vollständig unifizierte Term steht am Heap und beginnt bei Adresse 6 (siehe Tabelle 5.6). Die Zelle 2 verweist nun nicht mehr auf sich selbst, ist also gebunden worden. Da es sich bei den beiden Vorkommnissen der Variable X um dieselbe Variable handelt, verweist Zelle 5 auf Zelle 2. Weil die Variable an den Term a/0 gebunden ist, verweist Zelle 2 auf Zelle 10, ab der der Term a/0 durch zwei Zellen repräsentiert wird.

Nach der Unifikation steht der Term p(f(a),g(a)) am Heap. Die Adresse der Variable X ist 2 (bzw. 5) und wir erhalten daher als Ergebnis der Unfikation: X=a.

Argumente von Prädikaten werden in Prolog immer mit denen der aufgerufenen Prozedur unifiziert. Daher ist die Unifikation eines der grundlegenden Elemente Prologs. Im Folgenden wird das Abarbeiten von Prozeduren und Backtracking erläutert.

5.1.4 Prozeduren

Wie in imperative Sprachen, ist die Abarbeitungsreihenfolge von links nach rechts. Die Ziele in Klausen werden wie Prozeduren aufgerufen. Vor Aufruf einer Prozedur müssen die Daten, die durch die Seiteneffekte der 'put'-Befehle zerstört werden könnten, in einem Environment zwischengespeichert werden.

E	CE (Stackadresse des vorigen Environments)
E+1	CP (Adresse des auszuführenden Befehls bei Rückkehr aus einer Prozedur)
E+2	n (Anzahl der permanenten Variablen)
E+3	Y1 (permanente Variable 1)
	$\vdots$
E+n+2	Yn (permanente Variable n)

Tabelle 5.7: Environment

Ein Environment enthält eine Teilmenge der lokalen Variablen einer Klause. Diese werden in der WAM-Terminologie permanente Variablen genannt (siehe 5.2). Permanente Variablen werden also im Speicher abgelegt. Weiters enthält ein Environment die Adresse des nächsten Befehls, der bei erfolgreicher Rückkehr aus einer Prozedur auszuführen ist. Die Adresse entspricht dem Inhalt des globalen Registers CP bei Erzeugung des Environments auf dem lokalen Stack. Environments werden auf einem Stack gemeinsam mit den in 5.1.5 erläuterten Wahlpunkten als verkettete Liste abgelegt. Es ist auch ein Verweis auf das vorige Environment vorhanden. Dieser Verweis ist nötig, da die Größe eines Environments von der Anzahl der Variablen abhängig ist, die in diesem zwischengespeichert werden. Die Anzahl der permanenten Variablen ist daher ein wichtiger Faktor bei Effizienzüberlegungen und Optimierungen.

Bei Aufruf eines Prädikates (bzw. einer Prozedur) werden permanente Variablen und das globale Register CP in einem Environment gespeichert.

Für die Klause $p_0(\ldots)\leftarrow p_1(\ldots),\ldots,p_n(\ldots)$ werden die Befehle gemäß folgendem Schema erzeugt:

p_0: allocate N
Argumente von p_0 mittels get-Befehlen unifizieren
Argumente von p_1 mittels put-Befehlen schreiben
call p_1
$\vdots$
Argumente von p_n mittels put-Befehlen schreiben
call p_n
deallocate

Die Befehle 'allocate' und 'deallocate' erzeugen bzw. entfernen Environments am lokalen Stack. Der Parameter N gibt die Anzahl permanenter Variablen an, die im neuen Environment gespeichert werden. Das Speichern von Variablen wird in der WAM immer von der aufrufenden Prozedur übernommen (engl. *caller saved*). Die 'get'-Befehle unifizieren Register im Kopf einer Klause, 'put'-Befehle laden vor Aufruf einer Prozedur die Argumente in die Register bzw. schreiben Datenstrukturen auf den Heap. Ein Prädikat wird durch den Befehl call aufgerufen.

5.1.5 Backtracking

Das Rücksetzen der Berechnung ist eine Eigenschaft von Prolog, die in imperativen Sprachen nicht zu finden ist. Es müssen Variablenbindungen aufgelöst werden, Environments freigegeben und die nächste Klause eines Prädikates ausgewählt werden. Für Prädikate mit mehr als einer Klause muß durch Ausführung des WAM-Codes eine Datenstruktur erzeugt werden, die die Auswahl der Klausen bei Backtracking festlegt. Diese Datenstruktur heißt *Wahlpunkt*.

Das logische ODER ist in Prolog durch mehreren Klausen in einem Prädikat realisiert. Wird nun eine dieser Klausen bearbeitet, so muß der Ausführungsstatus der WAM zwischengespeichert werden, um bei Scheitern dieser Klause im Programmablauf fortsetzen zu können. WAM-Befehle haben Seiteneffekte auf Stacks und globale Register. Um diese wieder rückgängig zu machen, werden die dafür notwendigen Daten in einem Wahlpunkt gespeichert.

Durch Verwendung eines Wahlpunktes kann also der Zustand der WAM, den sie vor Ausführung der Klause eines Prädikates hatte, wiederhergestellt werden. Ein Wahlpunkt enthält folgende Daten:

- Argumentregister der Prozedur (die Anzahl der Argumentregister entspricht der Arität des Prädikates). Diese müssen gespeichert werden, da Argumentregister durch 'put'-Befehle beim Aufruf von anderen Prädikaten überschrieben werden können.
- Das aktuelle Environment. Mit diesem Environment wird im Falle von Backtracking fortgesetzt.
- Die Programmadresse, an der fortgesetzt wird.
- Der vorige Wahlpunkt. Dieser wird benötigt, wenn alle Alternativen des aktuellen Wahlpunktes scheitern.
- Die nächste Klause, die bei Scheitern der aktuellen Klause verwendet wird, um fortzusetzen.
- Die Stelle des Trail, von der aus Variablenbindungen im Falle von Backtracking aufgehoben werden müssen.
- Das Ende des Heaps, um Speicher, der während der Ausführung eines gescheiterten Prädikates verwendet worden ist, freizugeben und wieder verwenden zu können.

Es existieren drei Befehle, um Wahlpunkte am lokalen Stack zu erzeugen, zu modifizieren und zu beseitigen (L ist das Label eines WAM-Befehls im Code-Bereich):

try_me_else L: Am Stack wird ein neuer Wahlpunkt erzeugt. In diesem wird ein Verweis auf die nächste alternativen Klause mit L initialisiert. Weiters wird die Arität des Prädikates und die Argumentregister gespeichert. Die globalen Register H, E, CP sowie TR werden ebenfalls im neuen Wahlpunkt gespeichert.

retry_me_else L: Dieser Befehl wird für die Klausen zwischen der ersten und der letzten Klause verwendet. Es werden sämtliche Variablenbindungen zurückgenommen, die seit der Erzeugung des Wahlpunkts am lokalen Stack erzeugt worden sind. Die globalen Register H, E, CP und TR werden mit den Werten im Wahlpunkt restauriert. Das Feld für die nächste alternative Klause wird mit L initialisiert.

trust_me: Die globalen Register werden wie bei retry_me_else restauriert und der Wahlpunkt vom lokalen Stack entfernt. Das Register B wird auf den vorigen Wahlpunkt gesetzt.

Diese Befehle werden im Code für ein Prädikat folgendermaßen verwendet:

p/n	:	try_me_else L_1
		Code für die erste Klause
L_1	:	retry_me_else L_2
		Code für die zweite Klause
		$\vdots$
L_{k-1}	:	retry_me_else L_k
		Code für die vorletzte Klause
L_k	:	trust_me
		Code für die letzte Klause

Enthält ein Prädikat nur zwei Klausen, so werden nur die Befehle try_me_else und trust_me verwendet. Besteht ein Prädikat nur aus einer einzigen Klause, so ist keiner dieser drei Befehle nötig, da auch kein Wahlpunkt erzeugt werden muß.

5.1.6 Speichermodell

Der Zustand der WAM wird in sechs verschiedenen logischen Bereichen verwaltet. Zwei Stacks, der globale und der lokale Stack, werden für Datenobjekte verwendet, ein Stack (PDL), um Unifikation zu unterstützen, der Trail Stack, um das Zusammenwirken von Unifikation und Backtracking zu unterstützen. Weiters wird ein Bereich für das Programm (code) und ein Bereich für die Verwaltung von Symbolen verwendet.

Heap

Dieser Stack beinhaltet Listen und Strukturen. Der Heap wird schematisch als Stack mit adressierbaren Zellen betrachtet. Eine einzelne Zelle besteht aus zwei Feldern, einer Kennzeichnung und einem Wert. Ist die Kennzeichnung STR oder REF so ist der Wert eine Adresse, ist sie Funktor, so ist der Wert das Funktionssymbol und dessen Arität. Terme werden durch diese Zellen am Heap repräsentiert. Für den Funktor eines Terms wird eine Struktur-Zelle (STR) und eine Funktor-Zelle am Heap erzeugt. Dann folgen die einzelnen Argumenten, wobei Argumente selbst wieder Terme sein können. Ungebundene Variablen werden durch Zellen repräsentiert, die auf sich selbst verweisen.

Lokaler Stack

Dieser Stack beinhaltet Environments und Wahlpunkte. In Environments werden lokale Variablen gespeichert. Die Anordnung dieser beiden Strukturen am Stack spiegelt die Gültigkeitsbereiche der einzelnen Klausen wieder. Die permanenten Variablen, die am lokalen Stack gespeichert werden, werden auch Stack-Variablen genannt.

Trail

Am Trail-Stack werden die Adressen der Variablen, die gebunden werden, gespeichert. Terminiert eine einzelne Regel einer Klause zu False, so müssen alle Variablen, die im Laufe der Abarbeitung dieser Regel gebunden worden sind, wieder frei gegeben werden, d.h., wieder als freie Variablen bei Abarbeitung der nächsten Klause zur Verfügung stehen. Das Speichern der Adressen der Variablen heißt *Trailing*, das Auflösen der Bindungen *Untrailing*. Aber nicht alle Variablen, die gebunden werden, müssen auch am Trail gespeichert werden. Die Adresse einer Variable muß nur dann gespeichert werden, wenn sie auch nach Backtracking weiterhin benötigt wird. Dies kann durch die sogenannte Trail-Bedingung geprüft werden. Wenn die Adresse einer Variable am lokalen oder globalen Stack kleiner ist, als die des obersten Elements des Trails, dessen Adresse im jüngsten Wahlpunkt gespeichert ist, dann muß diese Variable auch noch nach erfolgtem Backtracking existieren.

Push-down Liste (PDL)

Diese dynamische Datenstruktur wird verwendet, um während der Unifikation die Adressen von Termen zu verwalten.

Code-Bereich

Der Code-Bereich ist ein adressierbares Array von Datenworten. Jedes dieser Worte enthält ein Befehlswort gefolgt von Operanden. Dem Befehlswort kann zusätzlich noch ein Label hinzugefügt sein, das als symbolische Adresse auch als Operand von Befehlen benutzt werden kann. Dieser Bereich wird auch als permanenter Speicher bezeichnet, da die Daten während des gesamten Programmablaufes hier nicht verändert werden. Ausnahmen bilden die Befehle assert und retract, die Klausen hinzufügen und entfernen.

Symboltabelle

Wie auch in anderen Programmiersprachen, müssen Namen von Prozeduren, Variablen und Strukturen verwaltet werden.

5.2 Variablenklassifikation

Variablen, die vor und nach einem Prozeduraufruf vorkommen, müssen vor den Seiteneffekten der 'put'-Befehle geschützt werden. Man unterscheidet zwei Arten von Variablen. Ist es nötig, eine Variable am lokalen Stack zwischenzuspeichern, so spricht man von einer permanenten Variable, andernfalls heißt sie temporäre Variable. Warren's Definition ist wie folgt:

> Eine temporäre Variable tritt in höchstens einem Ziel auf, wobei der Kopf der Klause als Teil des ersten Ziels gesehen wird, und sie tritt zuerst im Kopf, in einer Struktur, oder im letzten Ziel auf. Alle anderen Variablen heißen permanente Variablen [War77].

Folgendes Beispiel verdeutlicht diese Unterscheidung:

p(X,Y) $\leftarrow$r(X,Z),s(Z,Y).

X ist eine temporäre Variable, da sie nur in einem Ziel vorkommt und das erste Mal im Kopf auftritt. Y und Z sind permanente Variablen. Da der Kopf zum ersten Ziel gerechnet wird, tritt Y in zwei Zielen auf und ist daher eine permanente Variable. Ebenso tritt Z in zwei Zielen auf. Wäre der Zugriff auf die Register Y und Z nur von einem Argumentregister aus möglich, so könnte nicht sichergestellt sein, daß sie auch nach der Unifikation von X noch die richtigen Werte enthalten. Daher werden die permanenten Variablen Y und Z in einem Environment am lokalen Stack gespeichert. Eine etwas andere Definition wird in [AK91] angegeben. Durch diese Definition werden weniger Variablen als temporär klassifiziert, um die Verwendung von Heapspeicher zu minimieren. Es wird aber nicht in allen Fällen effizienterer Code erzeugt.

5.3 Optimierungen

Durch die in Kapitel 4 vorgestellten Kontinuationstransformationen können folgende Optimierungen erreicht werden:

1. Die Anzahl der Argumente eines Prädikates wird verringert.

2. Die Anzahl der permanenten Variablen in einem Prädikat wird verringert.

3. Werte verbleiben über Prädikatsgrenzen hinweg im selben Register.

Wird die Anzahl der Argumente eines Prädikates verringert, werden auch weniger Befehle benötigt, um die Argumente auf den Heap zu schreiben und zu unifizieren. Die in 5.1.3 erläuterten Befehle zur Behandlung von Argumenten können somit reduziert werden, wodurch eine Effizienzsteigerung erreicht wird.

Die Verringerung der Anzahl der permanenten Variablen in einem Prädikat hat mehrfache Auswirkungen auf die Datenstrukturen der WAM. Die Größe eines Environments (siehe Tabelle 5.7 auf Seite 69) ist direkt abhängig von der Anzahl der permanenten Variablen, die in diesem gespeichert werden. Da in jedem Wahlpunkt auch das aktuelle Environment gespeichert wird, kann auch die Größe von Wahlpunkten verringert werden.

Eine Trail-Überprüfung umfaßt all jene Variablenbindungen, die seit dem letzten Wahlpunkt durchgeführt wurden, um sie im Falle von Backtracking wieder freigeben zu können. Bei Listendifferenzen wird bereits „zu Beginn“ eine Variable reserviert, die erst später gebunden wird. Da durch Kontinuationstransformationen eben diese Variable beseitigt wird, entfallen alle mit dieser Variable in Zusammenhang stehenden Trail-Überprüfungen.

5.4 Vergleich

Wir besprechen nun die Auswirkungen von Kontinuationstransformationen auf die Code-Erzeugung von SICStus Prolog. In 4.5.3 optimierten wir das Prädikat ausdruck/3. Wir vergleichen nun den WAM-Code, der für ausdruck/3 erzeugt wird, mit dem Code für das optimierte Prädikat ausdruck1/2. Wir gehen vom ursprünglichen Prologprogramm aus und geben dann jeweils an, welche WAM-Befehle durch die Optimierungen beseitigt oder durch andere ersetzt wurden.

Betrachten wir nochmals das Prädikat ausdruck/3:

ausdruck(b(W),[b(W)|Xs],~~Xs~~).
ausdruck(knoten(Bl,Br),[op|Xs0],~~Xs~~) ←
    ausdruck(Bl,Xs0,~~Xs1~~),
    ausdruck(Br,Xs1,~~Xs~~).

Das dritte Argument, das durch Kontinuationstransformationen beseitigt wird, ist jeweils durchgestrichen. Die permanenten Variablen Xs1 und Xs, die optimiert werden, sind unterwellt. Wir geben nun den WAM-Code für das ursprüngliche Prädikat ausdruck/3 und das optimierte Prädikat ausdruck1/2 an. Die Interface-Regel ist hier nicht angegeben, da diese durch die Ersetzung aller Aufrufe von ausdruck/3 durch ausdruck1/2 in einem Programm beseitigt werden kann.

Prolog	WAM-Code	optimierter WAM-Code	Low-Level Prolog
ausdruck(	try_me_else L1	try_me_else L1	ausdruck1(
b(	get_structure_x0(b/1)	get_structure_x0(b/1)	b(
W),	unify_x_variable(3)	unify_x_variable(3)	W),
[	get_list(1)	get_list(1)	[
|Xs],	unify_temp_variable(4)	unify_temp_variable(4)	|Xs]):-
Xs	unify_x_local_value(2)	unify_x_variable(2)	
b(	get_temp_structure(b/1,4)	get_temp_structure(b/1,4)	b(
W)	unify_x_value(3)	unify_x_value(3)	W)
).		put_x_value(2,1)	'SINK'(–,+Xs).
	proceed	proceed	
ausdruck(	L1: trust_me	L1: trust_me	ausdruck1(
k(	get_structure_x0(k/2)	get_structure_x0(k/2)	k(
Bl,	unify_x_variable(0)	unify_x_variable(0)	Bl,
	allocate	allocate	
Br),	unify_y_variable(0)	unify_y_variable(0)	Br),
[	get_list(1)	get_list(1)	[
op	unify_constant(op)	unify_constant(op)	op
Xs	get_y_variable(2,2)		
|Xs0]	unify_x_variable(1)	unify_x_variable(1)	|Xs0]
)	put_y_variable(1,2)		)
:-	init([])	init([])	:-
ausdruck(Bl,Xs0,Xs1),	call(ausdruck/3,3)	call(ausdruck/3,3)	ausdruck1(Bl,Xs0),
	put_y_value(0,0)	put_y_value(0,0)	'SOURCE'(–,+Xs1),
	put_y_unsafe_value(1,1)		
	put_y_value(2,2)		
	deallocate	deallocate	
ausdruck(Br,Xs1,Xs).	execute(ausdruck/3)	execute(ausdruck/3)	ausdruck1(Br,Xs1).

In den ersten beiden Spalten ist das ursprüngliche Prologprogramm und der von SICStus Prolog erzeugte WAM-Code zu sehen. In den beiden anderen Spalten die optimierte Version. Die Schreibweise des SICStus WAM-Codes ist etwas anders als die in 5.1.3 verwendete. Aus Effizienzgründen existieren in SICStus Prolog verschiedene Befehle für die X und Y Register. Daher wird nur die Registernummer als Parameter angegeben. So wird etwa statt unify_variable X1 der Befehl unify_x_variable(1) verwendet.

Die erste Klause von ausdruck/3 ist ein Faktum. Die Variablen W und Xs sind temporäre Variablen und konnten daher nicht optimiert werden. Der Code für die Unifikation der ersten beiden Argumente ist daher gleich. Unterschiedlich ist der Code für die Behandlung der Variable Xs. Diese stand im dritten Argument von ausdruck/3 und wurde beseitigt. Der Code für das Faktum unterscheidet sich in folgenden zwei Befehlen:

1. unify_x_local_value(2) wurde durch unify_x_variable(2) ersetzt. Der wesentliche Unterschied zwischen diesen Befehlen ist, daß der ursprüngliche Befehl die Variable X2 mit einer Zelle am Heap oder am lokalen Stack unifizierte, der neue Befehle die Variable X2 für die Unifikation aber nur vorbereitet. Durch unify_x_variable(2) wird im Lese-Modus dem Register X2 der Inhalt jener Zelle am Heap zugewiesen, auf die S verweist. Im Schreib-Modus wird eine ungebunden Zelle am Heap erzeugt. Diese Operationen sind einfacher als die des ursprünglichen Befehls, durch den die Unifikation von Zellen durchgeführt wurde. Es mußten Zellen dereferenziert und Bindungen hergestellt werden.

2. put_x_value(2,1) ist in der optimierten Version hinzugekommen. Der Grund hierfür ist der Befehl 'SINK'(–,+Xs) im Low-Level Prologprogramm. Dieser bewirkt, daß sich nach Abarbeiten des Codes der Inhalt der Variable Xs im zweiten Argumentregister befindet. Durch put_x_value(2,1) wird der Inhalt des Registers X2 in das Argumentregister A1 geschrieben [1].

Im Code für die zweite Klause entfallen vier Befehle. Die Variablen Br, Xs1 und Xs waren gemäß der Klassifikation in 5.2 permanente Variablen. Sie wurden in den Registern Y0,Y1 und Y2 im Environment gespeichert. Durch Kontinuationstransformationen konnten wir ein Argument beseitigen und die Variablen Xs1 und Xs optimieren. Die Anzahl der permanenten Variablen wurde von 3 auf 1 verringert. Die Variablen Xs1 und Xs müssen daher nicht mehr in das Environment geschrieben werden. Befehle, durch die das Lesen und Schreiben in die Stack-Variablen Y1 und Y2 durchgeführt wurde, entfallen im optimierten Code. Dies hängt auch mit der Beseitigung des dritten Arguments zusammen, da Xs1 und Xs auch im dritten Argument auftraten. Es wurden folgende Optimierungen des WAM-Codes erreicht:

[1] Die Numerierung der Argumentregister beginnt in SICStus Prolog bei 0. Das zweite Argumentregister steht daher in A1.

1. Da das dritte Argument beseitigt wurde, muß das Argumentregister A2 nicht mehr für die Unifikation des Kopfes der Regel initialisiert werden. Daher ist der Befehl get_y_variable(2,2) im optimierten Programm nicht mehr vorhanden. Dadurch wird auch ein Zugriff auf die Stack-Variable Y2 beseitigt, da durch diesen Befehl Y2 dem Argumentregister A2 zugewiesen wurde.

2. Für den Aufruf von ausdruck(Bl,Xs0,Xs1) wurde im ursprünglichen Programm das Argumentregister A2 mit dem Inhalt von Y1 initialisiert. Auch in diesem Fall hängt die Beseitigung des dritten Arguments mit der Beseitigung von Befehlen für das Lesen von Y1 zusammen. Der Befehl put_y_variable(1,2) entfällt, da es nicht mehr nötig ist, den Inhalt der Stack-Variable Y1 in A2 zu schreiben.

3. Auch für den zweiten rekursiven Aufruf ist das Initialisieren des Argumentregisters A1 im optimierten Code nicht mehr nötig. Der Befehl put_y_unsafe_value(1,1) entfällt. Es ist nicht mehr nötig das Argumentregister A1 nach dem Aufruf von ausdruck1/2 für den zweiten rekursiven Aufruf mit dem Wert von Y1 zu initialisieren, da nach einem Aufruf von ausdruck1/2 das Argumentregister A1 diesen Wert bereits enthält. In der Stack-Variable Y1 wurde im ursprünglichen Programm die Variable Xs1 gespeichert. Xs1 mußte für den zweiten rekursiven Aufruf im Argumentregister A1 stehen. In Low-Level Prolog wurde durch die beiden Befehle 'SINK'(–,+Xs) und 'SOURCE'(–,+Xs) zugesichert, daß das Argumentregister A1 nach der Rückkehr aus der Prozedur den Wert des dritten Arguments enthält. Im Faktum wurde aus diesem Grund der Befehl put_x_value(2,1) benötigt, um dem Argumentregister A1 diesen Wert zuzuweisen. Durch diesen Befehl wird also der Wert von Xs1 von der ursprünglich dritten Argumentposition an die zweite geschrieben. Das entspricht dem Vorkommen der Variable Xs1 in den beiden rekursiven Aufrufen an dritter und zweiter Stelle.

4. Es verbleibt noch der zweite rekursive Aufruf von ausdruck1/2. Da das dritte Argument beseitigt wurde, ist es nicht mehr notwendig, das Argumentregister A2 für den Aufruf von ausdruck1/2 zu initialisieren. Der Befehl put_y_value(2,2) entfällt daher. Somit konnten alle Zugriffe auf die Stack-Variablen Y1 und Y2 beseitigt werden. Daher muß für diese kein Speicher mehr im Environment reserviert werden, wodurch die Größe des Wahlpunkts verkleinert wird.

Die permanente Variable Br konnte nicht optimiert werden. Diese wird nach wie vor am lokalen Stack in einem Environment gespeichert. Ein Environment wird durch den Befehl allocate erzeugt, der im Code der zweiten Klause an dritter Stelle auftritt. Der Befehl allocate wird bei der Code-Erzeugung immer so spät wie möglich erzeugt, da die Klause schon durch Befehle vor allocate scheitern könnte und in diesem Fall das Erzeugen des Environments entfallen kann. Die dritte Position ist optimal, da an vierter Position bereits die Stack-Variable Y0 für die permanente Variable Br benötigt wird.

Durch Beseitigen des dritten Arguments und Optimieren der zwei permanenten Variablen Xs1 und Xs konnten 4 Befehle in der rekursiven Regel eingespart werden.

Abschließend berechnen wir die Anzahl der Stack-Variablen, die bei der Ausführung des Prädikates nun nicht mehr im Environment gespeichert werden müssen. Die binären Bäume, die durch das Prädikat ausdruck/3 beschrieben werden, haben Knoten, die immer zwei Nachfolger haben. Der Nachfolger ist entweder wieder ein Knoten oder ein Blatt. Daher ist die Anzahl der Knoten immer um genau 1 kleiner als die Anzahl der Blätter. Für einen vollständigen Baum mit n Ebenen ist die Anzahl der Blätter gleich 2^{n-1}. Die Anzahl der Knoten ist gleich $2^{n-1} - 1$. Dieses Verhltnis gilt auch für unvollständige Bäume, allerdings ist die Anzahl der Ebenen dann ungleich n. Für jeden Knoten muß die zweite Klause einmal aufgerufen werden. Im ursprünglichen Prädikat mußten bei jedem Aufruf drei Stack-Variablen reserviert werden. Wir konnten zwei Stack-Variablen beseitigen. Da für das Einebnen eines (möglicherweise unvollständigen) Baumes mit 2^{n-1} Blättern die zweite Klause zur Bearbeitung der Knoten $2^{n-1} - 1$ mal aufgerufen wird und pro Aufruf zwei Stack-Variablen weniger reserviert werden, müssen insgesamt $2^n - 2$ weniger Stack-Variablen erzeugt werden.

Zusammenfassung

In diesem Kapitel wurde die Funktionsweise und der Aufbau der Warren Abstract Machine erläutert. Wir sahen, daß die WAM einen Befehlssatz und eine Speicherarchitektur enthält, um Prolog effizient zu implementieren. Die Begriffe Wahlpunkt und permanente Variable wurden eingehend besprochen. Wir stellten fest, daß die Größe eines Wahlpunktes von der Anzahl der permanenten Variablen, die in ihm gespeichert werden, abhängig ist und ein entscheidender Faktor für die Effizienz von Programmen ist. Weiters besprachen wir die Code-Erzeugung für eine Klause, sodaß Backtracking entsprechend durchgeführt werden kann. Wir sahen, in welchen Fällen Wahlpunkte erzeugt und beseitigt werden und welche Daten in diesen gespeichert werden. Die Unifikation wurde ausführlich an Hand einiger Beispiele erläutert und die Befehle vorgestellt. Daraus ging hervor, welche Befehle für die Bearbeitung von Argumenten eines Prädikates notwendig sind. Wir erläuterten die Trail-Bedingung und, daß Bindungen von Variablen, die im Trail-Stack gespeichert werden, im Falle von Backtracking wieder gelöst werden müssen. Abschließend wurde auf die Optimierungen eingegangen, die durch Kontinuationstransformationen erreicht werden können. Wir sahen, daß die Anzahl der permanenten Variablen verringert wird und, daß dadurch die Größe von Wahlpunkten verkleinert wird. Weiters wurde klar, daß die Verringerung der Anzahl der Argumente eines Prädikates eine Verringerung der für das Prädikat nötigen Befehle zur Folge hat. Abschließend haben wir die Auswirkungen von Kontinuationstransformationen auf die Code-Erzeugung untersucht, indem wir den WAM-Code eines Prädikates mit dem Code des optimierten Prädikats verglichen haben.

Durch die Darstellung der WAM war es möglich, die Auswirkungen von Kontinuationstransformationen auf die Code-Erzeugung in Begriffen der WAM zu erläutert.

Kapitel 6

Zusammenfassung

In dieser Diplomarbeit wurden verschiedene Möglichkeiten der Optimierung von logischen Programmen untersucht. Durch die in Kapitel 3 vorgestellten Techniken Falten/Entfalten konnte die rekursive Struktur von Programmen verändert werden. Dadurch konnten gemeinsame Variablen beseitigt werden. Auch konnten redundante Berechnungen beseitigt werden, wie sie beim mehrfachen Besuchen von Datenstrukturen auftreten. Weiters wurde gezeigt, daß durch Verwendung der in Kapitel 2 vorgestellten Listendifferenzen eine Effizienzsteigerung erreicht werden kann. Wir ersetzten normale Listen durch Listendifferenzen und konnten das Prädikat append/3 in Programmen beseitigen.

Durch die in Kapitel 4 umfassend dargestellten Kontinuationstransformationen konnten Listendifferenzen optimiert werden. Es stellte sich heraus, daß die existentielle Variable, die beim Durchreichen eines Differenzpaares entsteht, beseitigt werden kann. Weiters konnten Kontextargumente durch Generalisation in eine Form übergeführt werden, sodaß sie wie Listendifferenzen optimiert werden konnten. Das Einführen von Kontextargumenten erwies sich als geeignete Strategie, um Programme optimierbar zu machen. Dabei stellte sich heraus, daß die Position, an der Kontextargumente eingeführt werden, ein entscheidender Faktor für die Effizienzsteigerung ist. Das optimierte Programm konnte dann nach verschiedenen abstrakten Maschinen übersetzt werden. Um Code für die WAM zu erzeugen, wurde das Kontinuationsprologprogramm nach Low-Level Prolog übersetzt, das die zwei built-in Prädikate 'SOURCE'/n und 'SINK'/n enthielt, um die interprozedurale Registerallokation der WAM beeinflussen zu können. Durch die Übersetzung von Low-Level Prolog nach WAM-Code erhielten wir ein Programm, das Werte über Prädikatsgrenzen hinweg im selben Register halten konnte. Der Aufbau der WAM wurde in Kapitel 5 beschrieben. Die Optimierungen, die durch Kontinuationstransformationen erreicht werden, wurden nochmals untersucht. Es stellte sich heraus, daß die Verringerung der permanenten Variablen, die durch Kontinuationstransformationen erreicht wird, eine Verkleinerung der Environments und Wahlpunkte zur Folge hat. Weiters wurde die Übersetzung von Kontinuationsprolog nach binärem Prolog besprochen. Da binäres Prolog nur genau ein Ziel im Rumpf einer Klause aufweist, war es nötig, ein zusätzliches Argument und ein Hilfsprädikat einzuführen, um die Kontinuation lesen zu können. Binäres Prolog ist am besten für die Ausführung auf einer BinWAM geeignet. Die BinWAM unterscheidet sich in Hinsicht auf Aufrufkonventionen von der WAM, da jede Regel in binärem Prolog nur genau ein Ziel im Rumpf aufweist. Daher sind auch keine speziellen Mechanismen notwendig, um die Rückkehr aus einer Prozedur entsprechend zu handhaben, sondern es kann immer sofort die nächste Regel aufgerufen werden.

Die Semantik Prologs wurde in 2.3 und 3.3 untersucht. Es stellte sich heraus, daß die deklarative Semantik nicht immer gleich der prozeduralen Semantik ist, da es durch die Auswahlreihenfolge von Klausen eines Prädikates Endlosrekursionen auftreten können, durch die die Lösungsmenge verändert wird. Wir stellten auch fest, daß Kontinuationstransformationen die Sematik von Prologprogrammen erhalten.

In vielen Fällen führt die Anwendung verschiedener Transformationstechniken zum besten Ergebnis. So kann etwa durch Falten/Entfalten die rekursive Struktur eines Programms verändert werden und dann durch Kontinuationstransformationen einzelne Programmkonstrukte wie Listendifferenzen und Kontextargumente optimiert werden. Es stellte sich heraus, daß die bestmögliche Optimierung eines Programms durch Kombination verschiedener Strategien erreicht wird.

Literaturverzeichnis

[ADT95] J. Andrews, V. Dahl, P. Tarau. Continuation logic programming: Theory and Practice. ILPS 95 Workshop on Operational and Denotational Semantics of Logic Programs, Portland December 1995.

[AK91] H. Ait-Kaci Warren's Abstract Machine : A Tutorial Reconstruction. MIT Press, 1991.

[App92] A. Appel. Compiling with Continuations *Cambridge University Press*, 1992, ISBN 0-521-41695-7.

[BD77] R. M. Burstall und J. A. Darlington. Transformation System for Developing Recursive Programms. *Journal of ACM*, 24(1):44-67, 1977.

[BFS95] W. Binder, W. Faber und M. Schordan. Binäres Prolog und EBC-Transformation. Seminararbeit. Wintersemester 1994/95. Institut für Computersprachen, Technische Universität Wien, 1995.

[Deb85] S. K. Debray. Almost-Tail-Rekursive Prolog Programs. *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architekture*, Nancy, France, Lecture Notes in Computer Science 201, pages 204-219, Springer, New York, 1985.

[Deb88] S. K. Debray. Unfold/Fold Transformations and Loop Optimization of Logic Programms. *Proceedings of SIGPLAN '88. Conference on Programming Language Design and Implementation*, Atlanta, Georgia, SIGPLAN Not. 23(7):297-307, 1988.

[GB90] J. Gallager und M. Bruynooghe Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Metaprogramming in Logic*, pages 229-244. K. U. Leuven, Department of Computer Science, Apr. 1990.

[GH89] H-J. Goltz und H. Herre Grundlagen der logischen Programmierung Akademieverlag Berlin 1990. ISBN 3-05-500690-9

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. Proc. IFIP Congress, North Holland, Stockholm, 1974.

[Kow79] R. Kowalski. Algorithm = Logic + Control. In *Communications of the ACM* 22, pp. 424-436, 1979.

[Kow79b] R. Kowalski. Logic for problem solving. *Artificial Intelligence Series*, Volume 7, North Holland, New York, 1979.

[Llo87] J. W. Lloyd. Foundations of Logic Programming. 2. Auflage, Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo 1987.

[LT94] V. Lifschitz und H. Turner Splitting a Logic Program *11th ICLP 1994*, pages 23-36, MIT Press.

[Mei91] M. Meier Recursion vs. Iteration in Prolog. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 157-169, Paris, France, 1991. The MIT Press.

[Neu92a] U. Neumerkel. Specialization of Prolog programs with partially static goals and binarization, Dissertation. Bericht TR 1851-1992-12. Institut für Computersprachen, Technische Universität Wien, 1992.

[Neu92b] U. Neumerkel. Removal of existential variables by interprocedural register allocation in binary Prolog programs. Institut für Computersprachen, Technische Universität Wien, 1992.

[Neu93a] U. Neumerkel. A Transformation based on the equality between terms. LOPSTR, Springer-Verlag, 1993.

[Neu95] U. Neumerkel. Interprozedurale Registerallokation durch Quelltexttransformation. Institut für Computersprachen, Technische Universität Wien, 1995.

[Neu95b] U. Neumerkel Continuation Prolog: A new intermediary language for WAM and BinWAM code generation. ILPS 95 Workshop on Sequential Implementation Technologies for Logic Programming Languages, Portland December 1995.

[Neu96] U. Neumerkel Interprocedural register allocation for the WAM based on source-to-source transformations. JCSLP 95, MIT Press.

[Neu97] U. Neumerkel Continuation transformations for interprocedural register allocation. Institut für Computersprachen, Technische Universität Wien, 1997.

[Neu2] G. Neumann Meta-Programmierung und Prolog. Addison-Wesley, 1988. ISBN 3-925118-94-2.

[OKe90] R. A. O'Keefe. The Craft of Prolog. The MIT Press, Cambridge, Massachusetts, 1990.

[PP91] M. Proietti und A. Pettorossi. Unfolding - Definition - Folding, in this order, for avoiding unnecessary variables in logic programs. In J. Małuszyński and M. Wirsing, editors, *Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, Passau, Germany, Springer-Verlag, Aug. 1991.

[PP94] A. Pettorossi und M. Proietti. Transformation of logic programs: foundations and techniques. In *Journal of Logic Programming*, pp. 261-320, volume 19/20, 1994.

[PP94b] M. Proietti und A. Pettorossi Completeness of Some Transformation Strategies for Avoiding Unnecessary Logical Variables *11th ICLP 1994*, pages 714-729, MIT Press.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23-41, 1965.

[SS86] L. Sterling und E. Shapiro. The Art of Prolog. The MIT Press, Cambridge, Massachusetts, 1986.

[TB90] P. Tarau und M. Boyer. Elementary logic programs. In P. Deransart and J. Małuszyński, editors, *Programming Languages Implementation and Logic Programming*, volume 456 of *Lecture Notes in Computer Science*, Linköping, Sweden, Springer-Verlag, Aug. 1990.

[Tar91] P. Tarau. A simplified abstract machine for the execution of binary metaprograms. In *Proceedings of the Logic Programming Conference '91*, pages 119-128. ICOT, Tokyo, September 1991.

[TS84] H. Tamaki und T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the 2nd International Conference on Logic Programming*, pp. 127-138, Uppsala, Sweden, 1984.

[TS86] H. Tamaki und T. Sato. A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation, Technical Report No. 86-4, Ibaraki University, Japan, 1986.

[TS89a] T. Sato und H. Tamaki. First order compiler: a deterministic logic program. In *Journal of Symbolic Computation*, pp. 605-627, volume 8, Academic Press Limited, 1989.

[TS89b] H. Tamaki und T. Sato Existential continuation. *New Generation Computing*, 6(4):421–438, 1989.

[Wan80] M. Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164-180, 1980.

[War77] D. H. D. Warren. Implementing Prolog – Compiling Logic Programs 1 and 2. DAI Research Reports 39 and 40. University of Edinburgh, 1977.

[War83] D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA (October 1993)

[Yas95] R. Yasdi Logik und Programmieren in Logik. Prentice-Hall-Verl., 1995. ISBN 3-93043-624-8.

Danksagungen

Im Besonderen danke ich O.Univ. Prof. Dipl.-Ing. Dr. Manfred Brockhaus und Univ.Ass. Dipl.-Ing. Dr. Ulrich Neumerkel für die hervorragende Betreuung meiner Diplomarbeit. Durch Ulrich Neumerkel wurde mein Interesse an logischer Programmierung immer wieder um neue Aspekte erweitert, sodaß die Optimierung von logischen Programmen mir als besonders interessant erschien und mein Diplomarbeitsthema wurde. Weiters danke ich Paul Glaser, Walter Binder und Erwin Wechtl für die freundliche Hilfe beim Korrekturlesen.