DIPLOMARBEIT

Coroutining und Constraints in der Logik-Programmierung

ausgeführt am Institut für Praktische Informatik
der Technischen Universität Wien

unter Anleitung von
o. Prof. Dipl.-Ing. Dr. Manfred Brockhaus
und
Univ.-Ass. Dipl.-Ing. Dr. Andreas Krall

durch

Martin Anton Ertl

Wiedner Hauptstraße 141/1/7
1050 Wien

Wien, am 23. September 1992

Kurzfassung

Die Logik-Programmierung zeichnet sich durch einen deklarativen Programmierstil aus. Dadurch lassen sich Lösungen für Suchprobleme besonders leicht formulieren. Um diese Vorteile auch bei realen Problemen anwenden zu können, sind gegenüber Prolog noch einige Erweiterungen nötig.

Mit **Coroutining** können Zielaufrufe datengesteuert ausgeführt werden. Dies erleichtert die Trennung von Logik und Steuerung und damit die Verwendung eines deklarativen Programmierstils.

Constraint Logic Programming erweitert die auf syntaktischer Gleichheit beruhende Unifikation auf weitere Datentypen mit beliebigen Relationen. In diesem Rahmen können auch **Konsistenztechniken** in die Logik-Programmierung integriert werden. Konsistenztechniken ermöglichen die effiziente Lösung von kombinatorischen Problemen, indem sie den Suchbaum frühzeitig beschneiden und Informationen für die Anwendung von Heuristiken liefern.

Im Rahmen dieser Arbeit erfolgte die Implementation von Coroutining und Konsistenztechniken auf der Grundlage eines Prolog-Compilers für die Warren Abstract Machine. Die Einbindung in andere Prolog-Implementationen wird ebenfalls beschrieben.

Die Anwendungsmöglichkeiten von Logik-Programmiersprachen mit den besprochenen Erweiterungen werden an Beispielen aus Artificial Intelligence, Operations Research und Systemprogrammierung demonstriert.

Abstract

Logic Programming is distinguished by the declarative programming style it encourages. Solutions to search problems are specified easily. Some extensions to Prolog are necessary to apply these advantages to real-world problems.

Coroutining provides for data driven goal execution. Using Coroutining it is easier to separate logic and control and to use a declarative programming style.

Constraint Logic Programming liberates unification from its restriction to syntactic equality and extends it to relations on arbitrary computation domains. It also enables the integration of **consistency techniques** in logic programming. Combinatorial problems are solved efficiently using consistency techniques, since they prune the search tree early and provide information for the application of heuristics.

Coroutining and consistency techniques have been embedded in a Prolog compiler based on the Warren Abstract Machine. The application of these extensions to other Prolog implementations is described.

The application of the resulting extended logic programming languages is demonstrated using examples from artificial intelligence, operations research, and systems programming.

Inhaltsverzeichnis

Kapitel 1

Einführung

In vielen Anwendungen, vor allem in der künstlichen Intelligenz (Artificial Intelligence, AI) und im Operations Research, treten Suchprobleme auf, also Probleme, für die kein direkter Lösungsweg bekannt ist. Ähnlich der Suche in einem Irrgarten nach dem Ausgang muß sich ein Suchalgorithmus zunächst für einen bestimmten Lösungsansatz entscheiden, gleichzeitig aber alle anderen Ansätze offenhalten, um sie verwenden zu können, sollte der gewählte Ansatz nicht zum Erfolg führen.

Eine Unterklasse der Suchprobleme sind die CSPs (constraint satisfaction problems). Bei solchen Problemen müssen die Variablen einen Wert (label) aus einer endlichen Menge annehmen, wobei allerdings bestimmte Relationen (constraints) zwischen den Variablen beachtet werden müssen. Das Problem besteht darin, eine Belegung aller Variablen zu finden, die den Constraints genügt. Beispiele für CSPs sind Scheduling-Probleme und Graphenfärben im Operations Research, VLSI-Routing und -Layout in der Hardwareentwicklung, und Scene Labeling in der künstlichen Intelligenz (AI). Zu ihrer effizienten Lösung wurden die sogenannten Konsistenztechniken (consistency techniques) entwickelt.

Suchprobleme werden bisher mit den zur Verfügung stehenden allgemeinen Werkzeugen oder mit auf das spezifische Problem zugeschnittenen Spezialprogrammen gelöst.

Die allgemeinen Werkzeuge wie z.B. die Integer-Programmierung erfordern meist eine Umformulierung des Problems in die Begriffe des Werkzeugs und seiner Modelle. Diese Umformulierung ergibt meist ein komplizierteres und daher schwerer zu lösendes Modell, außerdem geht dadurch Information verloren, die die Suche abkürzen könnte. Wegen dieser Unzulänglichkeiten können schwierige Probleme oft nicht gelöst werden.

In einer imperativen Programmiersprache geschriebene Programme dagegen sind meist sehr effizient, erfordern aber viel Aufwand bei Entwicklung, Wartung und – hier besonders störend – bei Änderungen. Kleine Änderungen in der Problemstellung sind genauso schwer unterzubringen wie Änderungen der Suchstrategie, z.B. das Einbeziehen neuer Heuristiken.

Eine Alternative bietet die Logik-Programmierung, dank ihres Nondeterminismus, der dem Programmierer die Verwaltung der Suche abnimmt. Hinzu kommen noch die übrigen Vorteile der Logik-Programmierung, insbesondere der deklarative Programmierstil. Suchprobleme können wesentlich einfacher formuliert werden als in imperativen Sprachen.

Leider hat Prolog, die gebräuchlichste Logik-Programmiersprache, einige Unzulänglichkeiten, insbesondere eine starre und für Suchprobleme zu primitive Steuerungsstrategie und einen auf Herbrand-Terme beschränkten Berechnungsbereich. In einigen Logik-Programmiersprachen wurden diese Mängel allerdings behoben:

Coroutining erlaubt auf einfache Weise eine Steuerung der Beweisführung (Resolution), sodaß deklarativ geschriebene Programme effizient ausgeführt weden können.

Constraint Logic Programming (CLP) erlaubt die Erweiterung des Berechnungsbereichs, z.B. um Zahlen oder Strings und ihre Operatoren.

In diesem Rahmen können auch Konsistenztechniken in die Logik-Programmierung integriert werden. Dieser Ansatz ermöglicht eine effiziente Lösung von CSPs bei Beibehaltung der Vorteile der Logik-Programmierung. Da die Programme einfach und deklarativ geschrieben sind, können sie auch leicht an geänderte Problemstellungen angepaßt oder mit neuen Heuristiken versehen werden.

Coroutining und Konsistenztechniken wurden im Rahmen dieser Diplomarbeit in ARISTO eingebaut. Das Projekt ARISTO hat die Entwicklung einer industriellen CLP-Sprache zum Ziel.

Kapitel 2

Logik-Programmierung

2.1 Geschichte

Die Logik-Programmierung entwickelte sich aus den Bemühungen, die Prädikatenlogik zu automatisieren, um mechanisch Beweise führen zu können.

1965 gelang J. A. Robinson der entscheidende theoretische Durchbruch. Er entwickelte das Resolutionsprinzip, eine effiziente Methode, um Beweise über Horn-Klauseln zu führen. Horn-Klauseln sind ein Untermenge der Prädikatenlogik erster Ordnung und haben folgende Gestalt:

$$p \leftarrow q_1 \wedge q_2 \wedge \ldots$$

Kowalski zeigte 1974, daß Horn-Klauseln nicht nur als logische Aussage, sondern auch als operationale Anweisung gelesen werden kann:

> Um p zu beweisen, beweise q_1, beweise q_2, ...

Logik kann also als Programmiersprache aufgefaßt werden.

Auf der praktischen Seite wurde Anfang der siebziger Jahre in Marseille von einer Gruppe um Colmerauer eine solche Logik-Programmiersprache für den Einsatz in der Verarbeitung natürlicher Sprache entwickelt: Prolog. Durch die Entwicklung des DEC-10 Prolog-Compilers (1977) zeigte Warren, daß Prolog effizient verarbeitet werden kann und daß es sich durchaus mit Lisp messen kann.

2.2 Eigenschaften und Vorteile

Die Logik-Programmierung unterscheidet sich durch eine Reihe von Eigenschaften von imperativen Programmiersprachen. Dazu zählen referentielle Transparenz, symbolische Berechnung und Gleichungslösen durch Unifikation. Im Zusammenhang mit dieser Arbeit interessieren aber besonders der deklarative Programmierstil, die Invertierbarkeit von Programmen und der Indeterminismus zur Lösung von Suchproblemen.

2.2.1 Programmierstil

Die Logik-Programmierung ermöglicht einen deklarativen, problemorientierten Programmierstil. Das Problem wird beschrieben, nicht die Schritte zu seiner Lösung.

Beispiel: Die Ausgabe eines Sortier-Programms soll eine geordnete Permutation der Eingabe sein. Folgerichtig lautet ein Logik-Programm zum Sortieren (permutation sort, in Edinburgh-Prolog Syntax):

```
sort(L,Sorted):- permutation(L,Sorted), ordered(Sorted).
```

2.2.2 Invertierbarkeit

Logik-Programme kennen keinen Unterschied zwischen Ein- und Ausgabe-Argumenten. In der Tat können Prädikate mit verschiedenen Ein/Ausgabe-Mustern verwendet werden. Sind alle Argumente instantiiert, ist das Prädikat ein Test; sind ein oder mehrere Argumente noch frei, wird ihr Wert aus den anderen berechnet oder das Prädikat wirkt als Generator. Mit einem Programm kann ein Problem und sein inverses Problem gelöst werden. Diese Eigenschaft heißt Invertierbarkeit (invertibility).

Als Beispiel sei `append/3` genannt:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :-
        append(L1,L2,L3).
```

Es kann dazu verwendet werden, zwei Listen zu verketten, oder eine zu teilen:

```
?- append([1,2],[3],A).
```

ist erfolgreich mit der Ersetzung $\{A = [1,2,3]\}$, während

```
?- append(A,B,[1,2,3]).
```

gelingt und indeterministisch folgende Ersetzungen liefert:
$\{A = [], B = [1,2,3]\}$
$\{A = [1], B = [2,3]\}$
$\{A = [1,2], B = [3]\}$
$\{A = [1,2,3], B = []\}$

`append/3` kann auch in anderen Ein/Ausgabe-Modi verwendet werden, wenn z.B. alle Elemente Variablen sind, liefert es unendlich viele Lösungen:
$\{A = [], C = B\}$
$\{A = [D], C = [D|B]\}$
$\vdots$

2.2.3 Indeterminismus und Suche

Oft passen mehrere Klauseln auf ein Ziel. Jede dieser Klauseln kann verwendet werden, um das Ziel zu beweisen. Ein Ziel kann daher auch mehrere Lösungen haben. Die Klauseln können in einer bestimmten Reihenfolge oder parallel (Oder-Parallelismus) ausprobiert werden. Versagt eine Klausel, so kann immer noch eine Alternative Erfolg haben.

Aufgrund des Indeterminismus eignet sich die Logik-Programmierung ideal zum Lösen von Suchproblemen. Suchprobleme zeichnen sich dadurch aus, daß sie nicht durch einen normalen Algorithmus gelöst werden können, sondern daß verschiedene Möglichkeiten ausprobiert und überprüft werden müssen. An verschiedenen Stellen der Suche müssen Entscheidungen getroffen werden, die sich aber als falsch erweisen können. Daher müssen die Alternativen zu dieser Entscheidung immer offen gehalten werden.

Logik-Programmiersprachen führen eine Suche nach dem Beweis durch. Die Entscheidungen in der Suche entsprechen unmittelbar der Klauselauswahl in der Logik-Programmierung. Die Suche wird in der Logik-Programmierung vom System und nicht vom Benutzer verwaltet. Dadurch wird die Lösung solcher Probleme sehr erleichtert.

Auch Probleme, die normalerweise nicht als Suche gesehen werden, werden in der Logik-Programmierung oft einfacher als Suchprobleme formuliert. Als Beispiel kann hier wieder Permutation Sort herangezogen werden: Dieses Programm sucht eine Permutation, die geordnet ist.

Die Suche nach dem Beweis kann auch indeterministisch – vom Suchvorgang abstrahierend – gesehen werden: Das System wählt indeterministisch eine Klausel aus, die später zu einer Lösung führt.

2.3 Prolog

Prolog, die bekannteste Logik-Programmiersprache, wurde in den siebziger Jahren entwickelt. Beim Entwurf standen Einfachkeit, Effizienz, und allgemeine Verwendbarkeit im Vordergrund, was wohl den allgemeinen Erfolg bewirkt haben dürfte.

2.3.1 Berechnungsregel (computation rule)

Beim Entwurf einer Logik-Programmiersprache müssen zwei Entscheidungen getroffen werden:

- In welcher Reihenfolge werden noch zu beweisende Ziele abgearbeitet?
- Wie werden alternative Klauseln behandelt?

Prolog basiert auf der Tiefensuche (depth-first search) mit chronologischem Backtracking. Ziele werden in der Reihenfolge ihres Auftretens abgearbeitet, Klauseln werden in der Reihenfolge ihres Vorkommens ausprobiert. Diese Berechnungsregel ist zwar sowohl aus theoretischen Gründen als auch vom Standpunkt der Benutzbarkeit kritisiert worden [Kow85], ermöglicht aber eine effiziente Implementation. Im Kapitel 3 wird eine verbesserte Berechnungsregel besprochen.

2.3.2 Vordefinierte Prädikate

Um Prolog auch praktisch verwendbar zu machen, wurden sogenannte vordefinierte Prädikate (builtin predicates) eingebaut, die seine Schwächen etwas lindern.

Arithmetische Prädikate. In Prolog ist die Unifikation rein syntaktisch, d.h. Funktoren werden nicht interpretiert und daher versagt die Unifikation von `2+2` mit `4`. Um dennoch Arithmetik betreiben zu können, enthält Prolog entsprechende vordefinierte Prädikate. Allerdings gehen durch Beschränkungen bei ihrer Benutzung einige Vorteile der Logik-Programmierung verloren.

Metalogische Prädikate sprengen den Rahmen der Prädikatenlogik erster Ordnung und erlauben die Behandlung von Variablen als eigene Objekte statt als (noch unbestimmte) Terme. Das wohl wichtigste Beispiel ist `var/1`, das überprüft, ob sein Argument eine freie Variable ist. Metalogische Prädikate erlauben es, die durch die Berechnungsregel und andere Einschränkungen auftretenden Probleme zu handhaben.

Cut. Das Prädikat `!/0` steuert die Ableitung in einer für die Prolog-Berechnungsregel spezifischen und logisch nicht erklärbaren Weise. Die Anwendungen reichen von der Effizienzsteigerung bis zur Implementation der Negation, oft stimmt dann aber die deklarative Bedeutung eines Programms nicht mehr mit der prozeduralen überein.

Extralogische Prädikate, z.B. für die Ein- und Ausgabe, ergeben deklarativ gelesen keinen Sinn, haben also ausschließlich eine prozedurale Bedeutung.

Alle diese Prädikate sind ad-hoc-Lösungen bestimmter Probleme und passen schlecht bis überhaupt nicht in das Konzept der Logik-Programmierung. Für einige der motivierenden Probleme wurden inzwischen bessere Lösungen gefunden (siehe Kapitel 3 und 4).

2.3.3 Probleme

Die im Hinblick auf die praktische Verwendbarkeit eingegangenen Kompromisse werfen eine Reihe von Problemen auf, sodaß die Vorteile der Logik-Programmierung nur bedingt auch für Prolog gelten. Die starre Berechnungsregel führt zu gewissen Ineffizienzen bei der Suche nach der Lösung, zu Endlosschleifen, oder – zusammen mit den Einschränkungen bei vordefinierten Prädikaten – zu Laufzeitfehlern. Daher muß sich der Benutzer selbst um die prozedurale Ausführung der Programme kümmern; der deklarative Programmierstil geht verloren.

Ein für einen bestimmten Modus geschriebenes Programm funktioniert aufgrund der oben genannten Unzulänglichkeiten meist nur in diesem Modus. Daher geht auch die Invertierbarkeit verloren.

Auch Suchprobleme bereiten Schwierigkeiten, da Prolog mit seiner einfachen Berechnungsregel schnell der kombinatorischen Explosion erliegt.

Kapitel 3

Coroutining

Prologs einfache Berechnungsregel, die Ziele in einer Tiefensuche von links nach rechts abzuarbeiten, ist der Schlüssel zu einer effizienten Implementation und damit zur Brauchbarkeit als Programmiersprache. Andererseits gehen durch diese Regel viele Vorteile der Logik-Programmierung verloren: Programme können nicht mehr rein deklarativ geschrieben werden, da sonst Ineffizienz oder – bei rekursiven Prädikaten – Endlosrekursionen auftreten. Programmierer müssen daher Rücksicht auf die Steuerung nehmen und ihre Programme entsprechend gestalten. Das kann unter Umständen sehr schwierig sein, die Programme werden undurchschaubar.

Es fehlt daher nicht an Versuchen, die Steuerung von Prolog zu erweitern oder ganz abzuändern. Sowohl die Reihenfolge, in der die Klauseln eines Prädikats probiert werden, als auch die Reihenfolge der Zielaufrufe kann geändert bzw. zur Laufzeit bestimmt werden. Die dynamische Umordnung der Ziele zur Laufzeit heißt Coroutining – in Anlehnung an Mechanismen in imperativen Sprachen, die ähnlich angewendet und implementiert werden.

Ob ein Ziel keine, eine, mehrere, oder unendlich viele Lösungen liefert, sich in einer Endlosrekursion verfängt oder einen Laufzeitfehler liefert, hängt davon ab, inwieweit die Argumente des Ziels instantiiert sind. Daher überprüft der typische Coroutining-Mechanismus beim Aufruf eines Ziels die Argumente und verschiebt gegebenenfalls das Ziel, bis die Argumente ausreichend instantiiert sind.

Ein Beispiel ist ARISTOs Delay-Deklaration. Enthält ein Programm die Deklaration

```
:- delay p(nonvar, nonvar, any).
:- delay p(any, any, ground).
```

so wird bei jedem Aufruf von `p/3` überprüft, ob das erste und das zweite Argument instantiiert sind oder das dritte grundinstantiiert (ground) ist. Wenn ja, wird der Aufruf sofort ausgeführt, ansonsten wird er verschoben. Sobald die Bedingung durch eine Unifikation erfüllt ist, wird der Aufruf vor dem nächsten normalen Resolutionsschritt ausgeführt.

Die erste bekannte Logik-Programmiersprache mit Coroutining ist das am Imperial College in London entwickelte IC-Prolog [CMG82]. IC-Prolog ist der Versuch einer reinen Logik-Programmiersprache. Es enthält daher keine extra- und metalogischen Prädikate wie `!/0`, `var/1`, `assert/1` oder `retract/1`. Zum Ausgleich wird eine Anzahl von Steuerungsmechanismen zur Verfügung gestellt, unter anderem zwei Formen des Coroutining.

Prolog II [GKPvC86] wurde in Marseille entwickelt und enthält neben anderen Erweiterungen einen Coroutining-Mechanismus, und, darauf aufbauend, eine sichere (sound) Version der Ungleichheit.

Eine ähnliche Zielsetzung wie IC-Prolog verfolgt das an der Universität Melbourne entwickelte MU-Prolog [Nai86a], allerdings weniger puritanisch. Zu seinen hervorstechenden Merkmalen zählt ein Coroutining-Mechanismus und, darauf aufbauend, sichere Versionen der Negation, der Ungleichheit, und eines Alle-Lösungen-Prädikats (all solutions predicate, unsichere Versionen sind `findall/3` und `setof/3`). Ein Präprozessor für Prolog-Programme erzeugt automatisch Steuerungsinformation. MU-Prologs Nachfolger NU-Prolog enthält neben Quantoren und Negation [Nai86b] auch verfeinerte Steuerungsmöglichkeiten und wurde über einen Compiler auf Basis der WAM effizient implementiert.

3.1 Anwendungen

3.1.1 Programmierstil

Deklarativ geschriebene Prolog-Programme sind in der Ausführung, wenn sie überhaupt funktionieren, oft sehr ineffizient. Um brauchbare Programme zu erhalten, müssen die Benutzer daher Prologs Steuerungsmechanismus in Betracht ziehen, was die Les- und Wartbarkeit der Programme negativ beeinflußt. Teilweise muß sogar auf metalogische und extralogische Prädikate zurückgegriffen werden, sodaß die deklarative Bedeutung eines Programmes nicht mehr klar ist. In jedem Fall sind solche Programme schwieriger zu schreiben, zu verstehen und zu warten. Kowalski fordert daher eine Trennung von Logik und Steuerung [Kow79].

Steuerungsregeln wie Coroutining ermöglichen oft die effiziente Ausführung eines deklarativ geschriebenen Programms.

3.1.2 Invertierbarkeit

Eine der meistbeworbenen Eigenschaften der Logik-Programmierung ist die Invertierbarkeit der Programme, also die Möglichkeit, Prädikate je nach Instantiierungsmuster verschieden verwenden zu können.

Mit `append/3` können Listen nicht nur verkettet werden, eine Liste kann umgekehrt auch (indeterministisch) in zwei Teile geteilt werden. Aber schon bei ein bißchen komplizierteren Programmen geht diese Eigenschaft verloren. Das Prädikat

```
append3(A,B,C,D) :- append(A,B,X), append(X,C,D).
```

kann z.B. nur zum Verketten dreier Listen verwendet werden, beim Teilen gerät das Programm in eine Endlosrekursion. Die umgekehrte Anordnung der Unterziele hat das umgekehrte Verhalten zur Folge.

Das Verhalten von `append3/4` ist typisch für viele Prolog-Programme. Es kann, wie viele andere Programme auch, so umgeschrieben werden, daß es sich besser verhält, diese Aufgabe ist aber in vielen Fällen schwierig bis unmöglich. Programmierer sollten davon möglichst befreit werden.

Wenn jedoch der Aufruf von `append/3` verschoben wird, bis das erste oder das dritte Argument instantiiert sind, funktioniert `append3/4` sogar in Fällen, in denen jede der beiden Anordnungen zu Endlosschleifen führt (z.B. `append3([1|_],_,_,[2|_])`).

Als weiteres Beispiel sei das Prädikat `length/2` genannt, das sowohl zum Berechnen der Länge einer Liste als auch zum Erzeugen von Listen einer bestimmten Länge benutzt werden kann.

```
length([],0).
length([X|Y],N) :- N>0, plus(N1,1,N), length(Y, N1).
```

Aufgrund der üblichen Beschränkungen bei Aufrufen vordefinierter Prädikate kann `length/2`, so wie es geschrieben ist, in Prolog nur verwendet werden, wenn das zweite Argument instantiiert ist. Werden `>/2` und `plus/3` jedoch verschoben, bis sie genügend instantiiert sind, kann `length/2` in allen Modi verwendet werden, sogar mit zwei freien Argumenten, um indeterministisch Listen beliebiger Länge zu erzeugen.

3.1.3 Suchprobleme

Eine besonders einfache und elegante Art, Probleme zu formulieren und zu lösen, sind Generate&Test-Programme. Ein typisches Programm hat die Form:

```
find(X) :- generate(X), test(X).
```

Der Generator erzeugt über Backtracking Lösungskandidaten, im allgemeinen Lösungen eines leichteren Problems; der Tester überprüft die Kandidaten und läßt nur die Lösungen durch. Indeterministisch dargestellt: Der Generator erzeugt die Lösung und der Tester stellt sicher, daß sie auch wirklich eine Lösung ist.

Ein beliebtes Beispiel ist das N-Damen-Problem: N Damen müssen auf einem $N \times N$-Schachbrett so plaziert werden, daß sie sich nicht gegenseitig schlagen können.

```
queens(N,Qs) :- integers(N,Ns), permutation(Ns,Qs), safe(Qs).
```

Die Lösung wird als Liste der horizontalen Positionen der Damen dargestellt. Die vertikale Position entspricht der Position in der Liste. `permutation/2` erzeugt indeterministisch eine Permutation der Liste der Positionen (eine Lösung des N-Türme-Problems), `safe/1` überprüft, ob sich in dieser Stellung die Damen wirklich nicht schlagen können.

Generate&Test-Programme sind zwar leicht zu schreiben und leicht zu verstehen, aber hoffnungslos langsam, da der Generator meist sehr viele Kandidaten produziert, die keine Lösungen sind. Traditionellerweise wird daher der Tester mittels – meist händischer – Programmtransformationen (siehe Abschnitt 5.2.5) möglichst weit in den Generator hineinverlegt, um Versager möglichst früh zu eliminieren. Der Nachteil solcher Methoden ist der größere Arbeitsaufwand auf seiten des Programmierers, die Fehleranfälligkeit und die geringere Lesbarkeit des sich ergebenden Programmes. Mittels Coroutining können all diese Nachteile umgangen werden, das Programm muß nur leicht geändert werden:

```
queens(N,Qs) :- integers(N,Ns), safe(Qs), permutation(Ns,Qs).
```

Tests und ihre Teile können nicht vor dem Generator ausgeführt werden, weil sie bei ungenügender Instantiierung im allgemeinen selbst indeterministisch, also keine Tests, sind, oder – im Fall von vordefinierten Prädikaten – gar einen Fehler liefern. Daher müssen sie verschoben werden. Sobald aber innerhalb des Generators Instantiierungen erfolgen, können Tests ausgeführt werden, die eventuell versagen und dadurch das Versagen früh erkennen lassen. Coroutining hat in vielen Fällen die gleiche oder bessere Effizienz zur Folge wie Programmtransformationen.

Allerdings hat Coroutining auch Grenzen: Permutation Sort ist ein $O(n!)$-Algorithmus. Durch Coroutining läßt er sich nur auf $O(2^n)$ verbessern, während es bekanntlich Algorithmen mit der Komplexität $O(n \ln n)$ gibt.

3.1.4 Negation

In Prolog gibt es keine Klauseln mit negierten Köpfen. Somit ist Negation im Sinne eines Beweises, daß etwas falsch ist, unmöglich. Die Bedeutung der Negation in Prolog ist daher die Negation-as-Failure-Regel[1], das ist eine Vereinfachung der Closed World Assumption. Wenn ein Ziel (in endlicher Zeit) versagt, wird angenommen, daß seine Negation wahr ist.

Negation wird in Prolog üblicherweise wie folgt implementiert:

```
not(Goal) :- call(Goal), !, fail.
not(Goal).
```

Diese Definition funktioniert in einfachen Fällen, kann aber falsche Ergebnisse liefern, wenn `Goal` Variablen enthält. So versagt z.B.

```
?- not(X=1), X=2.
```

fälschlicherweise, ist aber bei umgekehrter Reihenfolge der Ziele erfolgreich. `not/1` funktioniert richtig, solange keine im negierten Ziel vorkommende Variable bei der Lösung des Ziels gebunden wird. Ein sicheres `not/1` ist am einfachsten zu erreichen, indem `not(Goal)` mittels eines Coroutining-Mechanismus verschoben wird, bis `Goal` grundinstantiiert ist:

```
:- delay not(ground).
```

Allerdings wird der Aufruf dadurch länger verzögert als nötig, z.B. braucht

```
not(member(1,[1|X]))
```

[1] Daher heißt das im folgenden als `not/1` bezeichnete Prädikat im Standard wahrscheinlich `fail_if/1`.

nicht verschoben werden, da das negierte Ziel in diesem Fall eine Lösung hat, die keine Variable bindet. Naish stellt in [Nai86a] andere sichere Lösungen der Problematik vor.

`\=/2` kann als Spezialfall von `not/1` verschoben werden, bis beide Argumente instantiiert sind. In diesem Fall ist es aber relativ einfach, das Ziel sobald wie möglich auszuführen, kann `X\=X` sofort versagen. Sowohl Prolog II als auch MU-Prolog enthalten entsprechende Prädikate.

`dif(X,Y)` in Prolog II z.B. ist wahr, wenn $X \neq Y$. Im Gegensatz zu `\=/2` in Standard-Prolog, das versagt, wenn die beiden Terme unifiziert werden können, ist `dif/2` sicher. Nur wenn die Unifikation gelingt, ohne eine Variable zu binden, versagt `dif/2`. Gelingt die Unifikation dagegen dank einer Variablenbindung, dann ist `dif/2` erfolgreich, das System merkt sich aber über den Coroutining-Mechanismus die noch nicht erledigten Ungleichungen und weckt sie wieder auf, wenn die Variablen gebunden werden. Der Aufruf

```
dif(f(X),f(Y))
```

ist z.B. zunächst erfolgreich, aber das System merkt sich $\{X \neq Y\}$.

3.1.5 Vordefinierte Prädikate

Viele von Prologs vordefinierten Prädikaten, z.B. die arithmetischen Prädikate wie `is/2` und `</2`, können nicht ausgeführt werden, wenn die Argumente noch zuviele freie Variablen enthalten. Konventionelles Prolog meldet in solchen Fällen einen Fehler. Durch Coroutining können solche Aufrufe so lange verschoben werden, bis die Argumente genügend instantiiert sind.

3.1.6 Werte-Fortpflanzung

Eine einfache Technik zur Lösung von Constraint-Systemen ist die Werte-Fortpflanzung (value propagation, local propagation).

Ein Beispiel bietet das vordefinierte Prädikat `plus/3`. `plus(A, B, C)` ist wahr, wenn $C = A + B$; Es wartet, bis zwei seiner Argumente instantiiert sind, und errechnet dann das übrige. Coroutining ermöglicht es, solche Constraints möglichst früh auszuführen und benutzt sie so aktiv, um neue Werte zu berechnen, anstatt die Werte durch Ausprobieren zu ermitteln.

3.2 Entwurfsfragen

3.2.1 Prädikats- oder aufrufbezogene Steuerung?

In IC-Prolog und Prolog II wird die Steuerungsinformation in den Aufrufen abgelegt. Prolog II's `freeze/2`-Prädikat ist wohl die bekannteste Form des Coroutining. `freeze(V, Goal)` verschiebt den Aufruf von `Goal` solange, bis `V` instanttiert wird. In IC-Prolog werden Variablen, die beim Aufruf instantiiert sein sollen, besonders gekennzeichnet.

Im Gegensatz zu IC-Prolog und Prolog II bezieht sich die Steuerungsinformation in MU-Prolog auf Prädikate und nicht auf Aufrufe. Für jedes Prädikat können Steuerungsdeklarationen gegeben werden, die dann für alle Aufrufe des Prädikats gelten. Diese Wait-Deklarationen haben folgendes Aussehen:

```
?- wait append(1,1,0).
?- wait append(0,1,1).
```

Dabei stehen die Einsen für Argumente, die bei einem Aufruf gebaut (constructed) werden dürfen. Ein Argument wird gebaut, wenn eine Variable im Argument instantiiert wird. Sind mehrere Wait-Deklarationen vorhanden, dann beschreiben sie verschiedene Arten, ein Prädikat aufzurufen. Die oben angeführte Deklaration verschiebt Aufrufe von `append/3`, bis das erste oder das dritte Argument im Aufruf nicht mehr gebaut werden muß.

Beide Formen sind gleich mächtig, sie können jeweils durch die andere ausgedrückt werden. Dabei ist es relativ einfach, `freeze/2` mit prädikatsbezogener Steuerung zu implementieren:

```
:- delay freeze(nonvar,any).
freeze(_,Goal):- call(Goal).
```

Umgekehrt muß immer ein Zwischenprädikat eingeführt werden:

```
member(X,L):- freeze(L,woken_member(X,L)).
woken_member(X,[X|T]).
woken_member(X,[Y|T]):- member(X,T).
```

Schwierigkeiten bereiten alternative Bedingungen wie bei `append/3`. Diese können aber mit folgendem Prädikat umgangen werden, das das Ziel verschiebt, bis eine der beiden Variablen instantiiert ist:

```
or_freeze(V1,V2,G):-
        freeze(V1,once(Called,G)),
        freeze(V2,once(Called,G)).

once(Called,G):- var(Called), Called=true, call(G).
once(Called,G):- nonvar(Called).
```

Die gemeinsame Variable `Called` stellt dabei sicher, daß das Ziel nur einmal aufgeweckt wird. Allerdings kann ein `once`-Ziel verschlafen, obwohl das eigentliche Ziel aufgewacht ist.

Da die beiden Mechanismen äquivalent sind, stellt sich die Frage, welcher verwendet werden soll. Das gewichtigste Argument für prädikatsbezogenes Coroutining ist die dadurch verwirklichte Trennung von Logik und Steuerung. Beim aufrufbezogenen Coroutining befindet sich die Steuerung mit der Logik vermischt in den Prädikaten und kann daher nur umständlich geändert werden. Auch das deklarative Lesen wird dadurch erschwert. Prädikatsbezogene Deklarationen sind dagegen relativ unabhängig von den zugehörigen Prädikaten und bereiten daher keine Mühe bei der Wartung und beim Lesen.

Für prädikatsbezogene Steuerung spricht weiters, daß Aufrufe desselben Prädikats meist dieselbe Steuerung bedingen, sodaß Steuerung beim Aufruf einiges an Redundanz verursacht. Für aufrufbezogene Steuerung spricht die einfachere Implementation (siehe Abschnitt 5.2.3).

3.2.2 Welche Art von Deklaration?

Nach der Entscheidung für prädikatsbezogenes Coroutining stellt sich die Frage nach der Art der Deklarationen. Dabei müssen die Einfachheit der Implementation, die Bedienungfreundlichkeit und die Ausdrucksmöglichkeiten gegeneinander abgewogen werden.

CHIPs Delay-Deklarationen sind sehr einfach. Es ist nur eine Deklaration der Form

$$\texttt{:- delay p(a}_1\texttt{, a}_2\texttt{, ...).}$$

erlaubt, wobei anstelle der $\texttt{a}_i$ `any`, `nonvar`, oder `ground` stehen darf. Aufrufe des Prädikats werden verschoben, bis alle Argumente die entsprechenden Bedingungen erfüllen.

Am anderen Ende des Spektrums befinden sich NU-Prologs When-Deklarationen.

```
?- append(A,B,C) when A or C.
```

verschiebt Aufrufe von `append/3`, bis das erste oder das letzte Argument instantiiert ist. Mit teilinstantiierten Termen anstelle der Argumente und durch Gebrauch mehrerer When-Deklarationen kann jede Bedingung dargestellt werden.

Die beste Steuerung gewähren When-Deklarationen. Andererseits ist ihre Implemenation beträchtlich aufwendiger als die einfacherer Mechanismen. In diesem Punkt schneidet CHIPs Delay-Deklaration am besten ab, dafür schränkt sie den Benutzer stark ein. Dadurch ist es z.B. sehr kompliziert, die für `append/3` gewünschte Steuerung auszudrücken.

Können aber mehrere Delay-Deklarationen für ein Prädikat verwendet werden, entfallen die schlimmsten Beschränkungen. Die Steuerung für `append/3` sieht jetzt so aus:

```
:- delay append(nonvar,any,any).
:- delay append(any,any,nonvar).
```

Kompliziertere Bedingungen sind selten notwendig und können gegebenenfalls durch Programmtransformationen ausgedrückt werden. Das Prädikat `ordered/1`, das feststellt, ob eine Liste geordnet ist, muß z.B. verschoben werden, bis sein Argument mindestens so instantiiert ist wie `[_,_|_]`. Ursprünglich lautet das Prädikat:

```
ordered([]).
ordered([A]).
ordered([A,B|T]):- A=<B, ordered([B|T]).
```

Die Bedingung `nonvar` reicht in diesem Fall nicht aus, `ground` dagegen wäre zu restriktiv. Die Lösung liegt in der Aufspaltung des Arguments:

```
:- delay ordered(nonvar).
ordered([]).
ordered([A|T]):- ordered(A,T).

:- delay ordered(any,nonvar).
ordered(A,[]).
ordered(A,[B|T]):- A=<B, ordered(B,T).
```

Mehrfache Delay-Deklarationen schränken also die Ausdrucksmöglichkeiten gegenüber When-Deklarationen nicht wesentlich ein, erlauben aber trotzdem eine einfache Implementation, ein wichtiges Kriterium beim Entwurf von ARISTO.

3.2.3 Verschlafen

Coroutining bringt nicht nur Vorteile, sondern auch neue Probleme. Es kann vorkommen, daß die Variablen, auf deren Instantiierung ein Ziel wartet, frei bleiben, und das Ziel daher nie aufgeweckt wird. Dieses Problem ist als Verschlafen (floundering) bekannt.

Ein Beweis, der mit schlafenden Zielen endet, ist weder erfolgreich, noch versagt er, da die Ausführung der schlafenden Ziele noch jeden Ausgang möglich macht. In IC-Prolog wird das Problem gelöst, indem die schlafenden Ziele aufgeweckt werden, wenn keine anderen mehr übrig sind. Da im allgemeinen die Ziele schlafen, weil sie sonst Endlosschleifen, unendlich viele Lösungen, oder andere Probleme erzeugen würden, ist diese Methode eher fragwürdig. Die meisten Systeme behandeln daher Verschlafen als Fehler und überlassen es dem Benutzer, das Programm oder die Anfrage so zu ändern, daß Ziele nicht verschlafen.

Das Problem des Verschlafens stellt sich auch bei Prädikaten, die unabhängig vom Rest des Programms vollständige Beweise führen müssen, wie `not/1` und Alle-Lösungen-Prädikate. Wenn z.B. der Aufruf von `Goal` mit Verschlafen endet, soll `not(Goal)` gelingen oder versagen? Auch hier ist Verschlafen als Fehler zu werten.

3.2.4 Reihenfolge des Aufweckens

Wenn mehrere Ziele gleichzeitig aufgeweckt werden, weil z.B. eine für mehrere Ziele verantwortliche Variable instantiiert wurde, stellt sich die Frage, in welcher Reihenfolge sie aufgeweckt werden. Die Reihenfolge wirkt sich folgendermaßen aus:

- Wenn die aufgeweckten Ziele Seiteneffekte haben, kann die Reihenfolge das Ergebnis bestimmen. Programme, die sich auf eine bestimmte Reihenfolge verlassen, zeigen aber einen schlechten Stil.
- Aus Effizienzgründen sollten Tests zuerst ausgeführt werden, dann deterministische Zielaufrufe, und schließlich die indeterministischen.

Die Reihenfolge kann durch den Benutzer spezifiziert oder vom System vorgegeben werden. Bei den mir bekannten Systemen beschränkt sich die Beeinflußung durch den Benutzer auf die Reihenfolge, in der die Ziele einschlafen. Im Sinne der Trennung von Logik und Steuerung wäre natürlich eine deklarative Form vorzuziehen, z.B. Prioritäts-Deklarationen. MU-Prolog und Prolog II rufen die Ziele in der Reihenfolge auf, in der sie einschliefen, IC-Prolog in der umgekehrten.

In ARISTO ist die Reihenfolge undefiniert, da die Vorteile einer definierten Reihenfolge die Komplikation der Implementation nicht aufwiegen. Die stiefmütterliche Behandlung dieser Problematik in der Literatur über die Implementation von Prolog II[Boi88] läßt allerdings den Verdacht aufkommen, daß auch dort die Reihenfolge des Einschlafens nur eingehalten wird, solange keine Komplikationen wie Unifikation zweier verantwortlicher Variablen oder Aufwecken zweier Ziellisten innerhalb eines Resolutionsschritts vorkommen.

3.2.5 Cut

Die Semantik von `!/0` muß neu definiert werden. Von der Implementation her ist es am einfachsten, wenn `!/0` auf alle Wahlpunkte wirkt, die seit dem Aufruf des aktuellen Prädikats erzeugt wurden. Von diesem Verhalten wären auch Wahlpunkte betroffen, die von aufgeweckten Zielen angelegt werden. Da es schwierig ist und außerdem dem Sinn des Coroutining widerspricht, vorherzusehen, wann ein Ziel aufgeweckt wird, verhalten sich Programme undurchschaubar, die sowohl Coroutining von indeterministischen Zielen als auch `!/0` benützen. Das Problem wird dadurch etwas entschärft, daß aufgeweckte Ziele meist Tests oder wenigstens deterministisch sind, also keine Wahlpunkte erzeugen, die einem Cut zum Opfer fallen könnten. Dieses Verhalten von `!/0` wird allgemein implementiert und außerdem verlassen sich einige Tricks darauf [Nai90]. Daher wurde es auch in ARISTO implementiert.

Von der Benutzerseite gesehen ist wohl ein Verhalten vorzuziehen, das die gleichen Lösungen liefert wie das ursprüngliche Programm. Dazu müßten allerdings auch eingeschlafene Ziele beschnitten werden. Es ist allerdings unmöglich, herauszufinden, wo ursprünglich geschnitten worden wäre bzw. wo der Benutzer den Schnitt gern gehabt hätte.

Daher muß man sich mit einer Lösung, die verständliches und durchschaubares Verhalten erzeugt, begnügen. Von diesem Standpunkt aus ist es vorzuziehen, wenn `!/0` alle verschiebbaren Ziele verschont. Diese Variante ähnelt einem Soft Cut. Abbildung 3.1 vergleicht die verschiedenen Verhalten.

Die Programme:

1.
```
:- delay member(nonvar,any).
p(X):- X=[1,2,3],!.
?- member(X,Y),p(Y).
```

2.
```
:- delay member(any,nonvar).
p(X,Y):- member(X,Y),!.
?- p(X,Y), Y=[1,2,3].
```

Das Verhalten von Cut:

1. Kein Coroutining
2. Cut wirkt auf alle Choicepoints, die seit dem Aufruf des aktuellen Prädikats erzeugt wurden.
3. Cut wirkt nicht auf die von aufgeweckten Zielen angelegten Choicepoints.

Verhalten	Programm 1	Programm 2
1	$X = 1$; $X = 2$; $X = 3$; Endlosschleife	$X = 1$; no
2	$X = 1$; no	$X = 1$; $X = 2$; $X = 3$; no
3	$X = 1$; $X = 2$; $X = 3$; no	$X = 1$; $X = 2$; $X = 3$; no

Abbildung 3.1: Die Wechselwirkung von Coroutining mit Cut

Kapitel 4

Konsistenztechniken

4.1 Constraint Logic Programming

Constraints sind Beziehungen zwischen Variablen. Beispiele:

- In der Schaltkreissimulation eines Widerstands sollten Strom, Spannung und Widerstand das Ohmsche Gesetz einhalten.
- Beim N-Damen-Problem dürfen zwei Damen nicht so stehen, daß sie sich gegenseitig schlagen können. Die Variablen sind dabei die Positionen der Damen.

Constraints dienen dazu, viele Probleme deklarativ zu formulieren. Die erste bekannte Arbeit ist Ivan Sutherlands Sketchpad-System, das in den frühen sechziger Jahren am MIT entwickelt wurde. Sketchpad erlaubte es den Benutzern, aus einfachen Graphik-Bestandteilen wie Linien und Kreisen mit Hilfe von Constraints Figuren zu erzeugen. Constraints werden oft in Simulationen eingesetzt, da die Modelle, auf denen die Simulationen beruhen, oft schon in Form von Constraints (als Gleichungen oder Ungleichungen) vorliegen, eignen sich aber auch für andere Probleme, von Benutzerschnittstellen bis zum Jazz-Komponieren.

Einen Überblick über Constraints, ihre Anwendungen und Lösungsmethoden vor allem außerhalb der Logik-Programmierung gibt [FBMB90].

Angesichts ihrer relationalen Form und ihres Indeterminismus ist die Logik-Programmierung eng mit Constraints verwandt. Es ist daher kein Wunder, daß eine Synthese dieser beiden Ansätze unternommen wurde. Sie erlaubt nicht nur die Lösung von Constraints im Rahmen der Logik-Programmierung, sondern ermöglicht es auch, viele Mängel der Logik-Programmierung zu beheben und ihre Vorteile zur Geltung zu bringen.

Eine theoretische Grundlage liefern Jaffar und Lassez mit ihrem CLP-Schema (constraint logic programming scheme) [JL87a, JL87b]. Wichtigstes Kriterium dabei war weniger die Formalisierung von Constraints innerhalb der Logik, sondern die Erhaltung der semantischen Eigenschaften der Logik-Programmierung.

Grundidee des CLP-Schemas ist es, die Unifikation der Herbrand-Terme durch einen Constraint-Löser für einen erweiterten Berechnungsbereich zu ersetzen. Wie müssen dazu Constraint-Löser und Berechnungsbereich beschaffen sein? Die Unifikation erfüllt zwei Aufgaben: Einerseits ist sie eine Entscheidungsprozedur (decision procedure), die uns sagt, ob eine Gleichung (über Herbrand-Terme) eine Lösung hat. Andererseits hat sie als weiteres Ergebnis eine Variablenersetzung, den sogenannten allgemeinsten Unifizierer (most general unifier, mgu), der die Lösung repräsentiert. Die Vereinfachung durch die Variablenersetzung ist aber nur eine Bequemlichkeit, ohne die die semantischen Eigenschaften genauso erhalten bleiben. Die Unifikation kann daher, vereinfacht gesagt, durch einen Constraint-Löser ersetzt werden, wenn dieser eine Entscheidungsprozedur ist, also für eine Menge von Constraints feststellen kann, ob sie eine Lösung haben oder nicht. Weiters muß der Berechnungsbereich lösungs-kompakt (solution compact) sein, was aber für alle in der Praxis vorkommenden Bereiche erfüllt ist.

Beispiele für in das CLP-Schema passende Sprachen:

Prolog mit dem Berechnungsbereich der Herbrandschen Terme und der normalen Unifikation als Constraint-Löser.

Prolog II aus Marseille [GKPvC86] mit zyklischen Termen (infinite rational trees) und einem Constraint-Löser für Gleichungen und Ungleichungen solcher Terme.

Prolog III, sein Nachfolger [Col87, Col90], der zusätzlich noch boolesche und lineare rationale Terme verarbeiten kann. Constraints auf Listen dienen als syntaktischer Zucker für Coroutining. Um ein effizientes System zu erhalten, in dem der Constraint-Löser eine Black Box ist, mußten allerdings starke Einschränkungen der Ausdrucksmöglichkeiten in Kauf genommen werden. Der Constraint-Löser für boolesche Terme beruht auf einer Sättigungsmethode, die mit SL-Resolution kombiniert wurde, rationale Gleichungen werden mit Gaußscher Elimination gelöst, für Ungleichungen wird eine inkrementelle Version des Simplex-Algorithmus verwendet.

CHIP (Constraint Handling in Prolog) [DVHS+88] wurde am ECRC (European Computer-Industry Research Centre) in München entwickelt und umfaßt neben Herbrand-Termen endliche Bereiche (finite domains), und ähnlich wie Prolog III boolesche [BS87] und lineare Terme [Gra87] sowie weitere Erweiterungen zur datengesteuerten Ausführung [Gra89]. Der Constraint-Löser für boolesche Terme beruht auf der booleschen Unifikation. Dadurch können wesentlich größere Probleme gelöst werden als mit Prolog III. Die Bereichs-Komponente paßt nach Ansicht von Van Hentenryck [VH89] nicht ins CLP-Schema, da die verwendeten Konsistenztechniken keine Entscheidungsprozedur sind. Gleiches gilt allerdings auch für die rationalen bzw. reellen Teile von CHIP und CLP($\Re$), sobald die Gleichungen nicht-linear werden.

Inzwischen wurden einige Sprachen auf der Grundlage des CLP-Schemas entwickelt, z.B.:

CLP($\Re$) [HJL+86] kann neben Herbrand-Termen auch noch lineare Gleichungen und Ungleichungen mit reellen Zahlen behandeln, ähnlich den rationalen Teilen von CHIP und Prolog III.

CLP(Σ^*) [Wal89] enthält einen Constraint-Löser für Zeichenketten, wobei die Constraints als reguläre Ausdrücke angegeben werden können. Dadurch können sowohl Zeichenketten als auch Text-Dateien in der Logik-Programmierung sauber behandelt werden.

Cosylog [BP89] ist eine Instanz des CLP(Conceptual Theory)-Schemas, das wiederum eine Spezialisiserung des CLP-Schemas für Wissensrepräsentationssprachen ist. In Cosylog gibt es „isa"-Constraints und Relationen, deren Eigenschaften (z.B. Symmetrie) vom Benutzer definiert werden können und die der Constraint-Fortpflanzung dienen. Cosylog wurde zum Verstehen natürlicher Sprachen eingesetzt.

Das CLP-Schema wurde auch schon erweitert: In [BMMW89] wird eine Erweiterung auf Constraint-Hierarchien beschrieben; von diesem HCLP-Schema gibt es auch schon die Instanz HCLP($\Re$, $\mathcal{LPB}$).

Der wohl wichtigste Beitrag des Schemas ist die Möglichkeit, die Logik-Programmierung auch auf nicht-Herbrandsche Daten zu erweitern, ohne daß sie wie bei den ad-hoc-Lösungen in Prolog ihre Vorteile verliert. Es bietet aber auch ein geistiges Gerüst zur Einbindung von Konsistenztechniken in die Logik-Programmierung.

Constraints können auch anders in die Logik-Programmierung integriert werden. Saraswat stellt die parallele Logik-Programmiersprache CP[$\downarrow$, $|$, &]

als Constraint-Sprache vor [Sar87]. CP[↓, |, &] bietet eine große Menge an Steuerungsmechanismen an, enthält aber keine für Constraints spezialisierten Teile. Constraints werden daher in CP[↓, |, &] formuliert und ohne eigenen Constraint-Löser gelöst. Die Effizienz soll dabei durch entsprechenden Einsatz der Steuerung gewährleistet werden.

4.2 Constraint Satisfaction Problems (CSPs)

Bei CSPs sind die Variablen auf endliche Bereiche eingeschränkt. Das Problem ist es, allen Variablen Werte zuzuweisen (ein labeling zu finden), sodaß alle Constraints erfüllt sind. Im allgemeinen sind CSPs NP-vollständige Suchprobleme, haben also (nach heutigem Wissensstand) exponentielle Komplexität.

Zur Lösung von CSPs bieten sich verschiedene Methoden, wie z.B. Truth-Maintenance-Systeme [McA90] an. Erfolgversprechend sind aber vor allem die Kombination von Konsistenztechniken (consistency techniques, relaxation algorithms, filtering algorithms) und Suchverfahren (tree search).

Am einfachsten zu implementieren sind reine Suchalgorithmen, die sich aber aufgrund der kombinatorischen Explosion meist nur für kleine Probleme eignen. Um die Laufzeit in erträglichen Grenzen zu halten, ist es wichtig, den Suchbaum frühzeitig zu beschneiden.

Im Rahmen der Konsistenztechniken [Mac77] hat jede Variable explizit einen Wertebereich. Die Werte werden auf ein bestimmtes Maß an Verträglichkeit mit den Constraints und den anderen Variablen überprüft und gegebenenfalls aus dem Wertebereich entfernt.

Beispielhaft sind im folgenden einige Suchalgorithmen angeführt, teilweise kombiniert mit Konsistenztechniken.

Generate & Test. Hierbei wird nach der Zuweisung von Werten an alle Variablen auf Einhaltung der Constraints geprüft. Ist die Kombination keine Lösung, wird die nächste probiert. Dieser Algorithmus ist natürlich sehr ineffizient, da er nicht einmal versucht, die kombinatorische Explosion in den Griff zu bekommen.

Standard-Backtracking überprüft nach jeder Zuweisung an eine Variable, ob dieser Wert mit den schon früher zugewiesenen Werten verträglich ist. Standard-Backtracking ermöglicht zwar eine gewaltige Steigerung gegenüber Generate&Test, leidet aber trotzdem an folgenden Mängeln:

- Dauerndes Wiederentdecken der gleichen Fakten in verschiedenen

Zweigen des Suchbaums.

- Versagen wird spät und oft erst nach weiteren – unnötigen – Zuweisungen erkannt.
- Backtracking zur letzten Zuweisung, die aber womöglich gar nichts mit dem Versagen zu tun hat. Als Folge wird langwierig ein Teilbaum durchsucht, der auf jeden Fall versagt.

Intelligentes Backtracking versucht, die Mängel des Standard-Backtrakking zu beheben, indem nicht die jüngste Zuweisung revidiert wird, sondern die jüngste für das Versagen verantwortliche und alle jüngeren Zuweisungen. Welche Zuweisung als schuldig erkannt wird, hängt vom jeweiligen Verfahren ab.

Forward Checking. Nach jeder Zuweisung werden aus den Bereichen der noch nicht zugewiesenen Variablen die Werte entfernt, die nicht mit der Zuweisung verträglich sind. Wenn der Bereich einer Variablen dadurch leer wird, gibt es im aktuellen Zweig des Suchbaums keine Lösung und Backtracking muß eingeleitet werden. Die Überprüfung auf Verträglichkeit mit den früher zugewiesenen Werten kann entfallen; ein unverträglicher Wert kann der Variablen ja nicht zugewiesen werden, da er schon vorher aus ihrem Wertebereich entfernt worden ist. Constraints werden also aktiv eingesetzt, sie beschneiden den Suchbaum a priori.

Lookahead. Zusätzlich zu den vom Forward Checking vorgenommenen führt Lookahead auch noch eine Überprüfung auf Verträglichkeit der noch nicht zugewiesenen Variablen durch, d.h. Werte, die sich bei jeder künftigen Zuweisung als unverträglich erweisen würden, werden eliminiert.

Haralick und Elliot [HE80] haben diese und einige weitere Algorithmen verglichen. Forward Checking erwies sich dabei als am effizientesten. Lookahead beschneidet den Suchbaum zwar noch stärker, kostet aber durch den stärkeren Aufwand beim Beschneiden insgesamt mehr Zeit. Den größeren Aufwand beim Lookahead schreiben Haralick und Elliot der Tatsache zu, daß es weniger inkrementell ist als Forward Checking, sich also die meisten Ergebnisse der zusätzlichen Prüfungen nicht merken kann und sie daher ständig wiederentdecken muß.

Die Effizienz der Suche kann auch durch die Reihenfolge beeinflußt werden, in der die Variablen instantiiert werden sollen. In Haralicks und Elliots Untersuchung erwies sich das Prinzip des schnellen Versagens (first fail principle) als erfolgreich: Nimm die Variable, die den kleinsten Wertebereich hat. Dadurch versagt die Suche schneller, wenn sie versagt, der Baum wird also früher beschnitten.

Eine weitere interessante Frage ist, welcher der möglichen Werte der Variablen zugewiesen werden soll. Wenn alle Lösungen gesucht werden, spielt das keine Rolle, da sowieso alle Werte des Bereichs durchsucht werden. Wenn aber nur eine Lösung gefunden werden soll, ist eine kluge Zuweisung wichtig. Dechter und Pearl [DP88] stellen einen Algorithmus vor, der ein dem zu lösenden Problem verwandtes leichteres (d.h. in polynomialer Zeit zu lösendes) Problem findet und löst und das dabei gewonnene Wissen dazu verwendet, um für die Zuweisung Ratschläge zu geben. Diese Ratschläge wirken zwar sehr gut, der Aufwand für den Algorithmus ist aber so groß, daß sich seine Verwendung bei den von Dechter und Pearl angeführten Beispielen nicht auszahlt. Einfache, problemspezifische Heuristiken dürften aber durchaus zu gebrauchen sein.

CSPs können auch ohne Suche gelöst werden: Freuder präsentiert in [Fre78] eine allgemeine Form der Konsistenztechniken, mit dem CSPs ganz ohne Suche gelöst werden können. In diesem Algorithmus werden die Constraints explizit dargestellt, also durch die erlaubten Wertekombinationen. Der Algorithmus erzeugt aus den vorhandenen neue Constraints von immer höherer Stelligkeit, bis schließlich ein Constraint ensteht, der alle Variablen erfaßt. Da im Constraint alle erlaubten Kombinationen explizit dargestellt sind, ist er auch die Lösung. Man kann diesen Algorithmus auch mit der Suche kombinieren. Unglücklicherweise ist er recht ineffizient.

4.3 Konsistenztechniken in der Logik-Programmierung

CSPs lassen sich in Prolog mit den Algorithmen Generate&Test oder Standard-Backtracking lösen. Um aber reale Probleme lösen zu können, ist der Einsatz effizienterer Techniken vonnöten. Van Hentenryck entwickelt in [VH89] den theoretischen und praktischen Rahmen zur Einbindung von Konsistenztechniken wie Forward Checking oder Lookahead in der Logik-Programmierung und gibt auch viele Anwendungsbeispiele. Er bezieht sich dabei zwar nicht auf das CLP-Schema, seine Arbeit kann aber auch in diesem Rahmen verstanden werden.

Da Konsistenztechniken darauf beruhen, daß sie die Wertebereiche der Variablen im voraus einschränken, müssen diese Bereiche in einem entsprechendem Datentyp abgespeichert werden, in den Bereichsvariablen. Diese Variablen können nicht beliebig instantiiert werden, sondern nur mit Werten ihres Bereichs. Versuche, Bereichsvariablen mit anderen Werten zu instantiieren, schlagen fehl. Bei der Unifikation von zwei Bereichsvariablen gelten für die sich ergebende, vereinigte Variable die Beschränkungen der beiden ursprünglichen Variablen, ihr Bereich ist also der Durchschnitt der beiden Bereiche. Sollte der Bereich einer Variablen nur mehr einen Wert enthalten, so wird die Variable mit diesem Wert instantiiert. Wird der Bereich leer, so kann die Variable nicht mehr instantiiert werden: die Ableitung versagt.

Bereichsvariablen erlauben nun die frühzeitige Elimination von Werten und damit eine Beschneidung des Suchbaums a priori, also bevor der Teilbaum erzeugt wird.

Nehmen wir als Beispiel die Ungleichung $X \neq Y$. Wenn `\=/2` sicher [Nai86a] implementiert ist (in Prolog ist das nicht der Fall), dann wird die Ungleichung erst dann ausgeführt, wenn beide Argumente grundinstantiiert sind oder wenn die beiden Variablen miteinander unifizieren. Wenn dagegen X eine Variable mit dem Bereich $\{1, 2, 3\}$ ist und $Y = 3$ ist, kann der Constraint sofort gelöst werden, indem der Bereich von X auf $\{1, 2\}$ eingeschränkt wird. Dadurch wird der Suchraum sofort kleiner, da Fälle, in denen $X = 3$ ist, nicht mehr berücksichtigt werden müssen.

Noch deutlicher ist die Reduktion des Suchraumes beim Constraint $X > Y$, wenn X im Bereich $\{1 \ldots 6\}$ liegt und Y im Bereich $\{4 \ldots 9\}$. X und Y können sofort auf die Bereiche $\{5, 6\}$ bzw. $\{4, 5\}$ eingeschränkt werden. Allerdings ist der Constraint dadurch noch nicht gelöst und muß weiterhin im Auge behalten werden – X könnte ja den Wert 5 annehmen, dann müßte Y mit 4 instantiiert werden.

Van Hentenryck integriert sowohl Forward Checking als auch Lookahead in die Logik-Programmierung. Welche Methode angewandt wird, ist vom Constraint abhängig. So ist z.B. $\neq$ ein Forward Checking Constraint, $>$ ein Lookahead. Constraints werden lokal angewandt, die gemeinsame Wirkung einer Menge von Constraints kommt nur insofern zum Tragen, als Variablenbereiche verändert und dadurch weitere Constraints angewendet werden.

Van Hentenryck definiert die Forward Checking Inference Rule (FCIR), die Lookahead Inference Rule (LAIR), und die Partial Lookahead Inference Rule (PLAIR) als theoretischen Rahmen für die Verwendung von Konsistenztechniken in der Logik-Programmierung. Die Schlußregeln stellen dabei eine Verallgemeinerung der in [HE80] für binäre Constraints angegebenen Algorithmen auf mehrstellige Constraints dar. Diese Schlußregeln bestimmen, wann und wie Constraints in der Ableitung angewendet werden können.

Gehorcht ein Constraint der FCIR, so kann er erst ausgeführt werden, wenn alle bis auf eine Variable grundinstantiiert sind und diese Variable eine Bereichsvariable ist. Der Constraint wird gelöst, indem aus dem Bereich der Variablen alle Werte entfernt werden, die den Constraint nicht erfüllen.

Nach der LAIR kann ein Constraint ausgeführt werden, sobald alle Argumente grundinstantiiert oder Bereichsvariablen sind. Dabei werden aus den Bereichen die Werte gestrichen, die mit keiner Kombination von Werten der anderen Variablen verträglich sind, die also bei jeder möglichen Instantiierung der Bereichsvariablen ein Versagen des Constraints auslösen. Der Constraint ist dann allerdings noch nicht erfüllt, und muß in der Ableitung bleiben, bis nur mehr eines seiner Argumente eine Bereichsvariable ist[1]. Dann kann er wie ein Forward Checking Constraint behandelt und als gelöst betrachtet werden.

Die PLAIR dient als Grundlage für die Implementation bestimmter vordefinierter Constraints. Das Grundprinzip ist ähnlich der LAIR, aber es müssen nicht alle unverträglichen Werte entfernt werden – ausgenommen der Constraint soll endgültig gelöst werden.

Schließlich muß die Berechnungsregel angepaßt werden. forward-effiziente Berechnungsregeln wählen Forward-Checking-Constraints aus, sobald die Bedingung zu ihrer Ausführung erfüllt ist. Die Behandlung von Lookahead Constraints ist etwas komplizierter, weil sie nach ihrem ersten Aufruf meist noch nicht gelöst sind. Sie dürfen daher erst wieder aufgerufen werden, wenn sie den Suchraum weiter einschränken können, wenn sich also ihre Argumente ändern. Berechnungsregeln, die dies berücksichtigen und aktuelle Lookahead-Constraints vor allen anderen Zielen wählen, heißen lookahead-effizient. Eine Berechnungsregel kann natürlich nicht gleichzeitig forward- und lookahead-effizient sein. Es ist allerdings leicht, eine effiziente Berechnungsregel für beide Arten von Constraints zu entwerfen.

CHIP

Bereichsvariablen werden in CHIP durch Domain-Deklarationen erzeugt:

```
domain p(h, 1..8).
```

bedeutet, daß der erste Parameter von `p/2` ein Herbrandscher Term ist, während der zweite eine Bereichsvariable mit dem Bereich $\{1, 2, \ldots, 8\}$ ist, mit der das hereinkommende Argument unifiziert wird. Diese Methode der Erzeugung ist allerdings nicht besonders elegant und widerspricht dem Prinzip, daß Deklarationen keinen Einfluß auf die Bedeutung eines Programms haben sollten.

[1] Genauer gesagt kann ein Lookahead dann als gelöst angesehen werden, wenn alle Kombinationen der Werte der Bereichsvariablen den Constraint erfüllen. Dies ist trivialerweise der Fall, wenn nur noch eine Bereichsvariable übrig ist und alle unverträglichen Werte aus dem Bereich der Variablen entfernt wurden.

Die Konsistenztechniken können auch für in Prolog definierte Constraints verwendet werden. Ein Prädikat `p/`n kann mit der Forward-Deklaration

```
forward p(a₁, ..., aₙ).
```

als Forward Checking Constraint deklariert werden. Die `a`$_i$ dürfen entweder `d` (für domain) oder `g` (für ground) sein. Die Deklaration bedeutet, daß das Prädikat nach der FCIR abgeleitet wird, sobald alle `g`-Argumente und alle bis auf ein `d`-Argument grundinstantiiert sind und das übrige Argument eine Bereichsvariable ist. Mit der Lookahead-Deklaration

```
lookahead p(a₁, ..., aₙ).
```

kann `p/`n als Lookahead Constraint deklariert werden.

In der Praxis sind vor allem die vordefinierten Prädikate von Bedeutung, die zwar vom Verständnis her genauso funktionieren wie die in CHIP definierten, aber wesentlich effizienter implementiert werden können.

Da durch Konsistenztechniken zwar die Wahlmöglichkeiten reduziert, aber nicht völlig eliminiert werden, stellt CHIP auch verschiedene Möglichkeiten zur Verfügung, durch Auswahl-Entscheidungen (choice) nach der Lösung zu suchen. Jede Entscheidung ermöglicht wiederum die Anwendung von Konsistenztechniken.

Das Prädikat `indomain/1` instantiiert die Bereichs-Variable indeterministisch mit einem ihrer Werte. Wenn die Instantiierung einer Variablen jedoch zu kostspielig ist, z.B. wenn es zu viele Alternativen gibt, stehen noch andere Mittel der Wahl zur Verfügung, z.B. die Bereichsteilung (domain splitting), bei der der Bereich einer Variablen in zwei Teile geteilt und die Variable indeterministisch auf einen Teil eingeschränkt wird.

Schließlich enthält CHIP noch einige Prädikate höherer Ordnung, die es erlauben, Optimierungsprobleme zu lösen. Sie beruhen auf einer Depth-First-Version des Branch-and-Bound-Algorithmus, die folgende Vorteile hat:

- Es werden früh gute Lösungen gefunden, sodaß es nicht so schlimm ist, wenn der Beweis der Optimalität wegen mangelnder Resourcen (z.B. Speicher) nicht gelingt.
- CHIP hat wie andere Logik-Programmiersprachen bereits eine Tiefensuche eingebaut, sodaß die Implementation der Suche entfallen kann.

Die Optimierungsprädikate erhalten ein Ziel, das alle Lösungen findet, und eine Kostenfunktion für die Lösungen, die zu optimieren ist. Zunächst wird eine Lösung ausgerechnet, und ihre Kosten c berechnet. Dann wird ein neuer Constraint erzeugt, der die weitere Suche einschränkt, z.B. daß die Kosten kleiner sein müssen als c. Dieser Constraint kann ebenso wie die Problem-Constraints aktiv verwendet werden, um die Suche zu beschränken. Auf diese Weise wird die Suche immer stärker eingeschränkt. Der Verzweigungsteil des Branch-and-Bound-Algorithmus wird also dem Benutzer überlassen, während die Kostenbegrenzung vom System übernommen wird.

CHIP enthält folgende Optimierungsprädikate und einige Verallgemeinerungen:

`minimize(Goal,Function).` `Goal` erzeugt die Lösungen. `Function` ist ein arithmetischer Term, der zu minimieren ist.

`minimize_maximum(Goal,List).` `List` ist dabei eine Liste von arithmetischen Ausdrücken, deren Maximum zu minimieren ist.

Ein Beispiel für die Anwendung dieser Prädikate ist das Graphenfärben, bei dem die Knoten eines Graphen mit der minimalen Anzahl χ von Farben so zu färben ist, daß nur verschiedenfarbige Knoten von einer Kannte verbunden werden. Im Programm werden die Farben durch Zahlen zwischen 0 und $\chi - 1$ dargestellt. Das Hauptprädikat eines hat folgendes Aussehen:

```
color_graph(Vertices):-
  state_constraints(Vertices),
  minimize_maximum(labeling(Vertices),Vertices).
```

Durch Minimieren der größten Zahl, die eine Kante hat, wird die Gesamtzahl der Farben minimiert. Im Optimierungsprädikat wird nur die Wertezuweisung ausgeführt, da sich die Constraints ja nicht ändern.

ARISTO

ARISTO unterscheidet sich in einigen Punkten von CHIP. So wurden die Domain-Deklarationen durch das Prädikat `in/2` ersetzt, das einer Liste von Variablen einen Bereich zuordnet. `[X] in 1..4` schränkt `X` auf den Bereich $\{1, 2, 3, 4\}$ ein.

Der wichtigste Grund für diese Entscheidung liegt in der mangelnden Flexibilität der Domain-Deklarationen. Es können keine Daten aus dem Programm einfließen, sodaß das N-Damen-Programm für jedes N verändert werden muß.

Außerdem beeinflußen Domain-Deklarationen die deklarative Bedeutung eines Programmes, die normalerweise den Prädikatdefinitionen (und nicht den Deklarationen) vorbehalten ist und es auch bleiben sollte.

`in/2` kann theoretisch mit Domain-Deklarationen definiert werden und ist somit semantisch genauso unbedenklich wie Domain-Deklarationen:

```
[] in D.
[0|T] in 0..0 :- T in 0..0.
[H|T] in 0..1 :- in_0_1(H), T in 0..1.
[H|T] in 0..2 :- in_0_2(H), T in 0..2.
⋮

domain in_0_1(0..1).
in_0_1(X).
domain in_0_2(0..2).
in_0_2(X).
⋮
```

Kapitel 5

Implementation

5.1 Prolog

5.1.1 Metainterpreter

Am häufigsten wird Prolog wohl in Prolog implementiert – als Metainterpreter. Diese Technik dient sowohl dazu, Debugger, Profiler und ähnliche Prolog-Entwicklungswerkzeuge als auch dazu, Prolog-ähnliche Sprachen zu implementieren. Metainterpreter sind sehr einfach, da für die meisten Dinge auf in Prolog eingebaute Mechanismen zurückgegriffen werden kann. Der in Prolog geschriebene Interpreter für CS-Prolog benötigt z.B. 1500 Zeilen, während das funktional ähnliche CLP($\Re$)-System als 12000-zeiliges C-Programm implementiert ist [KOM87].

Es müssen nur die Mechanismen implementiert werden, die verändert werden sollen. Im einfachsten Metainterpreter

```
solve(X):-call(X).
```

kann nichts verändert werden. Er ist daher relativ ungebräuchlich. Hingegen lassen sich die meisten Metainterpreter von folgendem Programm ableiten:

```
solve(X):-     sys(X), !, call(X).
solve((X,Y)):- !, solve(X), solve(Y).
solve(X):-     clause(X:-B), solve(B).
```

Metainterpreter sind naturgemäß langsam, was zu verschiedenen Versuchen führte, die Effizienz zu verbessern. Die meisten Versuche basieren auf der teilweisen Auswertung (partial evaluation) des Interpreters in Bezug auf das Programm. Im Gegensatz dazu wird in [Neu88] ein Compiler-Generator beschrieben, der aus dem Interpreter und ein bißchen Zusatzinformation einen Compiler erzeugt. Jener übersetzt dann für den Metainterpreter geschriebene Programme in Prolog-Programme, die auf einem Prolog-Interpreter um eine Größenordnung schneller laufen als ursprünglich. Bei einem Prolog-Compiler ist der Effizienzgewinn noch größer und steigt mit der Komplexität der Programme.

5.1.2 Interpreter

Bei einer Implementation in einer konventionellen, imperativen Programmiersprache müssen natürlich Prologs hervorstechende Mechanismen explizit implementiert werden. Dazu zählen insbesondere die Parameterübergabe mittels Unifikation und das Backtracking, aber auch die Erzeugung, Beseitigung und dynamische Typenprüfung bei Daten.

Darstellung von Objekten

Da die verschiedenen Arten von Daten in Form von Bitmustern vorliegen und nicht unterschieden werden können, werden sie mit Etiketten (tags) versehen, die bestimmen, wie sie zu behandeln sind. Variablen werden durch Speicherstellen – genannt Wertezellen (value cells) – dargestellt, die

- den Wert frei,
- einen Zeiger (Referenz) auf eine andere Wertezelle (entstanden aus der Unifikation der zweier Variablen),
- ein Atom, oder
- einen Zeiger auf eine Struktur

enthalten können.

Strukturen können auf zwei Arten dargestellt werden (siehe Abbildung 5.1). Beim Structure Copying [Mel82] werden Strukturen – ähnlich wie in imperativen Sprachen – als Speicherblock dargestellt, bestehend aus dem Funktor und den Strukturargumenten. Beim Structure Sharing dagegen bestehen Strukturen aus einem für alle Instanzen der Struktur gleichen Grundgerüst und einem Variablenblock, der die veränderlichen Teile der Struktur enthält. Beim Structure Sharing erspart man sich zwar bei jeder Struktur das erneute Abspeichern

Abbildung 5.1: Darstellung von Strukturen

der gleich bleibenden Teile (des Gerüsts), dafür muß in jeder Variable Platz für zwei Zeiger sein. Es gibt Fälle, in denen eine Methode wesentlich effizienter ist als die andere. Aber im allgemeinen benötigt Structure Copying gleich viel Speicher und etwas (15%) weniger Zeit [Kra88].

Unabhängig von der Darstellung der Datenstrukturen wird das Programm gewöhnlich mittels Structure Sharing dargestellt. Beim Ablauf gibt es dann einen Zeiger auf die aktuelle Stelle im Programm und einen auf die zugehörigen Variablen, das Environment.

Bei Interpretern wird das Programm meist genauso repräsentiert wie die Strukturen, was die Implementation von Meta-Prädikaten wie `call/1` und `clause/1` sehr vereinfacht.

Lebensdauer von Variablen

Im Gegensatz zu den meisten Algol-ähnlichen Sprachen können in Prolog Variablen einer Prozedur die Rückkehr vom Prozeduraufruf überleben, nämlich als Teil einer Struktur. Diese Variablen werden im globalen Stapel (global stack, bei Structure Copying Systemen auch copy stack) gespeichert und erst beim Backtracking oder bei der Speicherbereinigung (garbage collection) freigegeben. Die anderen Variablen können schon bei Verlassen der Prozedur oder noch früher freigegeben werden. Sie werden im lokalen Stapel (environment stack) gespeichert. Um ungültige Zeiger (dangling references) zu vermeiden, müssen Zeiger immer von Variablen mit kürzerer auf solche mit längerer Lebensdauer zeigen, also vom lokalen auf den globalen Stapel bzw. innerhalb des lokalen Stapels von der Spitze zur Wurzel.

Backtracking

Jedesmal, wenn mehrere Klauseln zur Auswahl stehen, muß ein Wahlpunkt (Choicepoint) eingeführt werden, zu dem gegebenenfalls zurückgekehrt werden muß. Bei Versagen müssen alle Änderungen rückgängig gemacht werden. Dies ist in Prolog sehr einfach: Jede Variable, der etwas zugewiesen wird, war vorher frei. Daher muß bei jeder Zuweisung nur die Adresse der Variablen auf einem eigenen Stapel, dem Trail, gespeichert werden. Beim Zurücksetzen werden diese Variablen wieder auf frei gesetzt. Weiters wird der lokale Stapel unterhalb eines Wahlpunkts nicht abgebaut - er könnte wieder benötigt werden.

5.1.3 Die Warren Abstract Machine (WAM)

Um kürzere Ausführungszeiten zu erreichen, wurden linearisierte Zwischencodes für abstrakte Maschinen entwickelt. Prolog-Programme werden von einem Compiler in einen dieser Zwischencodes übersetzt und dann direkt von einem Emulator interpretiert oder in Maschinencode für einen realen Prozessor weiterübersetzt.

Der bekannteste Zwischencode ist Warrens abstrakte Maschine [War83, GLLO85, Pic85]. Die WAM verwendet Structure Copying. Neben den Spezialregistern (Stapel-Zeiger etc.) enthält sie auch noch eine Anzahl von Argumentregistern, in denen bei einem Zielaufruf die Argumente übergeben werden. Die WAM hat folgende Befehlskategorien:

put -Befehle laden Argumente vor dem Unterzielaufrufs in Argumentregister und dienen so zur Parameter-Übergabe an Unterziele.

get -Befehle werden für den Kopf der Klausel erzeugt und unifizieren ein Argumentregister mit dem jeweiligen Argument im Kopf.

unify -Befehle dienen im Klauselkopf zur Unifikation von Strukturen, im Rumpf zum Erzeugen von Strukturen.

Steuerungs -Befehle dienen zum Aufrufen von Unterzielen etc. In diese Kategorie fallen auch die Befehle `call` und `execute` zum Aufruf eines Unterziels.

Indizierungs -Befehle zur Auswahl einer Klausel und Anlegen von Wahlpunkten.

Ein Beispiel ist in Abbildung 5.2 zu finden.

5.1.4 Die Vienna Abstract Machine (VAM)

Für das VIP-Prolog-System entwickelte Andreas Krall eine neue, effizientere Interpretationstechnik, die VAM [Kra88, KN90]. Die Grundidee ist, mittels zweier Befehlszähler die Argumente im Zielaufruf und im Klauselkopf zusammen zu interpretieren. Die Unifikation erfolgt direkt, ohne Umweg über die Argumentregister, was Zeit und durch zusätzlich verfügbare Information auch Speicher spart. Eine Variation, die VAM_{1P}, benötigt nur einen Befehlszähler und kann für die Weiterübersetzung in Maschinencode verwendet werden.

5.2 Coroutining

5.2.1 Metainterpreter

Die Resolutionssteuerung wird häufig als Metainterpreter implementiert, als Beispiel sei nur Epilog [Pei84] genannt. Neumann demonstriert die Anwendung seines Compilergenerators unter anderem an einem ziel-aufschiebenden Interpreter [Neu88]. Dieser merkt sich die schlafenden Ziele in einer Liste und prüft sie bei jedem Ableitungsschritt. Wenn aufgeschobene Ziele ausführbar geworden sind, werden sie aus der Liste entfernt und ausgeführt. Diese Vorgangsweise ist zwar nicht sehr effizient, aber einfach – der Metainterpreter besteht aus fünf Klauseln und drei Hilfsprädikaten. Die Übersetzung der Testprogramme mit dem aus dem Metainterpreter generierten Compiler ergab Beschleunigungen um den Faktor 1,4 bis 5.

Es ist natürlich wesentlich effizienter, für jede Variable eine Liste jener Zielen mitzuführen, die auf die Instantiierung dieser Variablen warten. Wird die Variable gebunden, so können ihre Ziele aufgeweckt werden, wenn sie nicht auf andere Variablen warten müssen. Um dieses Verfahren zu implementieren, ist es nötig, die Unifikation zu verändern. Daher muß im Metainterpreter auch die Unifikation explizit implementiert werden. Angesichts der Häufigkeit der

Prolog:

```
nrev([],[]).
nrev([X|L0],L):- nrev(L0,L1), append(L1,[X],L).
```

WAM-Code (mit einfachstem Indexing):

	`try_me`	`2, C2`		Wahlpunkt: versuche zunächst diese Klausel, dann die bei C2
	`get_nil`	`A0`	`nrev([],`	
	`get_nil`	`A1`	`[]`	
	`proceed`		`).`	return
`C2`	`trust_me`			dies ist die letzte Klausel, vergiß den Wahlpunkt
	`allocate`	`3`		lege Frame für 3 Variablen an
	`get_list`	`A0`	`nrev([`	
	`unify_localvar`	`Y0`	`X|`	speichere `X` in Variable 0
	`unify_regvar`	`A0`	`L0],`	`L0` lebt kürzer und kommt in ein Register
	`store`	`A1, Y1`	`L):-`	`L` ist Variable 1
	`put_local_var`	`Y2, A1`	`nrev(L0,L1`	`L0` ist schon im richtigen Register, `L1` kommt jetzt hinein
	`call`	`nrev`	`),`	Prozeduraufruf
	`put_unsafe_val`	`Y2, A0`	`append(L1,`	
	`put_list`	`A1`	`[`	Erzeuge Liste
	`unify_localval`	`Y0`	`X|`	mit Kopf `X`
	`unify_nil`		`[]],`	und Rest `[]`
	`load`	`A2, Y1`	`L`	
	`dealloc_execute`	`append`	`).`	Prozeduraufruf mit Tail-Call-Optimierung

Abbildung 5.2: Beispiel eines in WAM-Code übersetzten Prädikats (naïve reverse)

`unify(T1,T2)`	freie Variable	Atom/Struktur	Metastruktur
freie Variable	binde	binde	binde
Atom/Struktur	binde	vergleiche	`term_meta_unify/2`
Metastruktur	binde	`term_meta_unify/2`	`meta_meta_unify/2`

Abbildung 5.3: Unifikation, erweitert um Metastrukturen

Unifikation hat ihre Implementation in Prolog natürlich auch hohe Laufzeiten zur Folge.

Als Ausweg bieten sich Neumerkels Metastrukturen an [Neu90]. Metastrukturen sind ein neuer Datentyp, dessen Unifikationsalgorithmus vom Benutzer in Prolog definiert werden kann. Datentypen mit neuen Unifikationsregeln brauchen also nur als Metastrukturen definiert zu werden. Auf diese Weise erspart man sich die Implementation der normalen Unifikation und die damit einhergehenden Geschwindigkeitsverluste. Normale Prolog-Programme werden kaum langsamer.

Metastrukturen haben die Form `meta(Value,Property)`[1]. Sie verhalten sich normalerweise wie ihr `Value`-Teil, ist dieser aber frei, wird bei der Unifikation `term_meta_unify/2` bzw. `meta_meta_unify/2` aufgerufen, die dann mit `===/3` auf die Teile der Metastruktur zugreifen können[2]. Abbildung 5.3 zeigt die um Metastrukturen erweiterte Unifikationstabelle.

Mit Metastrukturen kann auch `freeze/2` implementiert werden [Neu90] (Abbildung 5.4). `freeze/2` hängt das Ziel als `Property`-Teil der Metastruktur an die Variable. Bei der Instantiierung wird das Ziel dann ausgeführt, während die Unifikation zweier verantwortlicher Variablen die beiden freien `Value`-Teile aneinanderbindet und ihnen dann die Konjunktion der Ziele anhängt.

5.2.2 Interpreter

Boizumault beschreibt in [Boi86] die Implementation von `freeze/2` auf Prolog-Interpretern sowohl mit Structure Sharing als auch mit Structure Copying.

Die Grundidee ist es, bei der Ausführung von `freeze(Var, Ziel)` `Ziel` direkt an `Var` zu binden. So kann das Ziel sehr einfach aufgeweckt werden, wenn die Variable instantiiert wird, also genau dann, wenn es ausgeführt werden soll. An jeder Variable hängt also eine Liste von Zielen, für die sie verantwortlich

[1] In der Praxis wird natürlich eine allgemeinere und praktischere Syntax verwendet.
[2] Die normale Unifikation würde den rekursiven Aufruf der Prädikate bewirken.

```
freeze(Variable,Goal) :-
    Variable = meta(_,Goal).

term_meta_unify(Term,Meta):-
   Meta === meta(Term, Goal),  % hole Ziel, binde Value-Teil
   call(Goal).                 % Zielaufruf

meta_meta_unify(Meta1,Meta2):-
   Meta1 === meta(Var,Goal1),
   Meta2 === meta(Var,Goal2),  % binde Value-Teile aneinander
   freeze(Var, (Goal1,Goal2)). % verantwortlich fuer beide Ziele
```

Abbildung 5.4: Implementation von `freeze/2` mit Metastrukturen

ist. Wenn zwei verantwortliche Variablen unifiziert werden, werden ihre Listen verkettet (genauer gesagt wird eine Struktur mit Zeigern auf die beiden Listen erzeugt). Ein schlafendes Ziel ist eine Prolog-Struktur und wird beim Aufwachen wie beim Meta-Call ausgeführt. Da jetzt Variablen vor dem Instantiieren nicht mehr nur frei sein können, wird ein Wertetrail (value trail) benötigt, der neben der Adresse auch den ursprünglichen Wert der Variablen speichert.

Um Speicherbereinigungstechniken ohne große Änderungen verwenden zu können, wird ein eigener Stapel für die Listen der schlafenden Ziele eingerichtet und die verantwortliche Variable global gemacht, sodaß keine lokale Variable auf den neuen Stapel zeigt. Als angenehmer Nebeneffekt braucht beim Unifizieren von verantwortlichen und wirklich freien Variablen nur eine Referenz auf die verantwortliche Variable in die Wertezelle der freien geschrieben werden. Der Stapel der schlafenden Ziele ermöglicht die Entdeckung von Floundering [Car87].

Coroutining ist in Prolog II ähnlich implementiert [VC86, Boi88]. Der Hauptunterschied liegt in der Speicherverwaltung. Außerdem weckt der eigentliche Coroutining-Mechanismus auch bei Bindungen an Variablen auf. Auf dieser Grundlage kann sowohl `freeze/2` als auch `dif/2` implementiert werden.

5.2.3 WAM

Prinzipiell läßt sich Boizumaults Modell auch auf die WAM anwenden, allerdings sind einige Besonderheiten zu beachten [Car87]: Vor dem Aufruf der aufgeweckten Ziele müssen die lebenden Register gesichert werden. Daher muß die Ausführung dieser Ziele verschoben werden, bis bekannt ist, welche Argumentregister noch leben. Andererseits sollen die aufgeweckten Ziele vor dem nächsten normalen Ziel ausgeführt werden. Carlsson wählte die Aufruf-Befehle `call` und `execute`, denen er offenbar Informationen über die Argumentregister zur Verfügung stellt.

Dadurch wird zwar die Ausführung der geweckten Ziele vor dem Aufruf des nächsten Prolog-Prädikates sichergestellt, jedoch können eingebaute Prädikate, die direkt in Zwischencode übersetzt werden, noch vorher ausgeführt werden. Carlsson verschiebt diese Ziele durch Decompilation: Warten aufgeweckte Ziele, wird das Prädikat nicht ausgeführt, sondern es hängt sich in Strukturdarstellung ans Ende der aufgeweckten Ziele.

Die Ausführung aufgeweckter Goals erfolgt in den Aufruf-Befehlen über einen allgemein verwendbaren Event-Handler-Mechanismus. Dabei werden zunächst die lebenden Register abgespeichert, die Rücksprungadresse wird auf den Befehl `continue` gesetzt, und schließlich werden die aufgeweckten Ziele mittels Meta-Call ausgeführt. Anschließend restauriert `continue` die Register und fährt mit dem ursprünglichen Programm fort.

Carlsson kann durch eine andere Verwaltung der verantwortlichen Variablen auch einen Wertetrail vermeiden (siehe Abbildung 5.5). Verantwortliche Variablen enthalten Zeiger (mit eigenem Etikett), die normalerweise wirken wie eine Referenz auf eine andere Wertezelle. Der Unterschied wird nur sichtbar, wenn die andere Wertezelle frei ist. Die nächste Zelle enthält dann nämlich einen Zeiger auf das schlafende Ziel. Soll nun eine Variable für ein Ziel verantwortlich gemacht werden, baut man eine Struktur mit zwei Feldern (Suspension) auf, wobei das erste Feld frei ist und das zweite auf das Ziel zeigt, und schreibt einen Zeiger auf die Struktur in das Wertefeld der verantwortlichen Variable. Da dabei nur ein freies Feld verändert wird, muß nur die Adresse getrailt werden. Dafür muß die Suspension-Struktur angelegt werden – ein etwas fragwürdiger Tausch. Auch die Laufzeit leidet unter diesem Schema: Bei jedem Zugriff auf eine verantwortliche Variable muß für jedes Ziel, für das die Variable verantwortlich ist, mindestens einmal dereferenziert werden.

Allerdings kann ein Teil der Kosten vermieden werden: Ist die Suspension jünger als der jüngste Wahlpunkt, so muß nicht getrailt werden. In diesem Fall kann die Suspension direkt geändert werden [Car90]. Bei der Speicherbereinigung können Dereferenzierungsketten verkürzt werden.

Die Suspension ähnelt der Implementation von `freeze/2` mit Metastrukturen.

Um Floundering zu entdecken, wird immer getrailt, wenn einer Variablen

Abbildung 5.5: Carlssons verantwortliche Variablen ein Suspension-Zeiger zugewiesen wird (ansonsten kann man sich das meistens ersparen). Durch Durchsuchen des Trails können dann alle schlafenden Ziele gefunden werden.

ARISTO

Insgesamt hat Carlsson das Coroutining scheinbar eher für die Nichtverwendung optimiert. Da jedoch in ARISTO Coroutining die Grundlage für die Bearbeitung von Constraints bilden und daher sehr intensiv genutzt werden, ist Boizumaults Schema – abgesehen von den von der WAM bedingten Abweichungen – geeigneter.

Weiters zwang die etwas andere Handhabung (Delay-Deklarationen statt `freeze/2`) zu einigen Änderungen. Dieses Problem könnte auch auf der Prolog-Ebene gehandhabt werden. Carlsson propagiert einen Präprozessor, der aus Coroutining-Deklarationen `freeze/2`-Aufrufe erzeugt [Car87]. Diese Methode bietet gegenüber der Implementation in einer maschinennahen Sprache keine Vorteile und ist weniger effizient und praktisch.

Die Delay-Deklarationen werden über einen eigenen WAM-Befehl implementiert, der vor dem jeweiligen Prädikat ausgeführt wird. Er überprüft dabei, ob eine Delay-Deklaration erfüllt ist. Wenn nicht, wird der Aufruf verschoben, wobei gegenüber `freeze/2` noch zwei Aufgaben hinzukommen: das Anlegen des schlafenden Zieles und die Auswahl der verantwortlichen Variablen.

Im Gegensatz zu `freeze/2` steht keine `call/1`-gerechte Struktur zur Verfügung, sodaß die Erzeugung eines schlafenden Zieles mehr Aufwand erfordert. Weiters muß die Definition der globalen Variablen um die Argumente von schlafenden Goals erweitert werden, da das Ziel aufwachen und damit auf die Variablen zugreifen kann, nachdem die Prozeduren beendet sind, die die Variablen erzeugt haben. Bei `freeze/2` waren die Variablen durch ihre Verwendung in der Ziel-Struktur automatisch global.

Die Auswahl der verantwortlichen Variablen hat zwei Ziele:

- Für jede Delay-Deklaration soll es (mindestens) eine verantwortliche Variable geben.
- Die Menge der verantwortlichen Variablen soll möglichst klein sein, um den Aufwand beim Verantwortlich-Machen und Instantiieren möglichst klein zu halten.

Dieses Problem ist isomorph zum Problem der Auswahl von Primimplikanten aus allen Primimplikanten einer logischen Funktion, sodaß die Funktion mit

---

Der Aufruf `p(f(A), g(B), C, [D], E, F, [G|H])` , wobei `p/7` folgende Delay-Deklarationen hat,

```
:- delay p(   any,    any,    any,    any, nonvar, ground, ground).
:- delay p(nonvar,    any,    any,    any, nonvar, nonvar, nonvar).
:- delay p(   any, nonvar,    any,    any, nonvar,    any, ground).
:- delay p(ground, nonvar,    any,    any, nonvar,    any,    any).
:- delay p(ground,    any, ground,    any,    any, ground, ground).
:- delay p(nonvar, nonvar,    any, nonvar,    any, nonvar,    any).
:- delay p(   any, ground,    any,    any,    any,    any, ground).
:- delay p(   any, ground,    any, ground,    any,    any,    any).
:- delay p(   any,    any, ground, ground,    any,    any,    any).
```

ergibt das folgende Muster von unerfüllten Delaydeklarationen:

unerfüllte Argument-spezifikationen		Delay-Deklaration 1	2	3	4	5	6	7	8	9
Argument	1				•	•				
	2							•	•	
	3					•				•
	4								•	•
	5	•	•	•	•					
	6	•	•			•	•			
	7	•		•		•		•		

Aus diesem Muster können nun die verantwortlichen Variablen gewonnen werden. Dazu konstruiert man sich eine Testfunktion $t(w_1, \ldots, w_7)$, die genau dann 1 ist, wenn die verantwortlichen Argumente ($w_i = 1 \Longleftrightarrow$ das i-te Argument ist verantwortlich) alle Delay-Deklarationen abdecken.

$$\begin{aligned} t &= (w_5 \vee w_6 \vee w_7) \wedge (w_5 \vee w_6) \wedge (w_5 \vee w_7) \wedge (w_1 \vee w_5) \wedge (w_1 \vee w_3) \wedge \\ &\quad (w_6 \vee w_7) \wedge w_6 \wedge (w_2 \vee w_7) \wedge (w_2 \vee w_4) \wedge (w_3 \vee w_4) \end{aligned}$$

Durch Anwendung des Quine-McClusky-Algorithmus erhält man alle Primimplikanten von t, nämlich $w_2 \wedge w_3 \wedge w_5 \wedge w_6$ und vier weitere, von denen man sich einen aussucht. Ein Primimplikant entspricht einer Menge von Argumenten, die verantwortlich gemacht werden müssen, in unserem Fall die Argumente 2,3,5 und 6, also die Variablen `B`, `C`, `E` und `F`.

Abbildung 5.6: optimale Auswahl der verantwortlichen Variablen den gewählten Primimplikanten konstruiert werden kann. Daher kann es mit dem gleichen, in [Pan86, Seiten 2-13f] angegebenen Algorithmus gelöst werden. Ein Beispiel ist in Abbildung 5.6 zu sehen.

Für die in der Praxis auftretenden Fälle genügt es jedoch völlig, wenn für jede Deklaration eine Variable ausgewählt wird und mehrfaches Auftreten einer Variablen eliminiert wird.

Wenn eine verantwortliche Variable instantiiert wird, werden alle Ziele, für die sie verantwortlich ist, überprüft und bei Erfüllung einer Deklaration aufgeweckt, ansonsten dürfen sie nach erneuter Bestimmung verantwortlicher Variablen weiterschlafen.

Um ein Verschieben eingebauter Prädikate zu vermeiden, steht vor ihrem Aufruf der neue WAM-Befehl `waken`, der aufgeweckte Ziele ausführt.

Die Entdeckung von Floundering erfolgt durch Zählen der eingeschlafenen und aufgeweckten Ziele.

5.2.4 VAM

Die Implementation von `freeze/2` ist einfacher als in der WAM, weil keine Rücksicht auf Argumentregister genommen werden muß. Sie kann direkt nach Boizumault durchgeführt werden. Andererseits erschwert die fehlende zeitliche Trennung zwischen Parameterübergabe und Unifikation die Implementation von Delay-Deklarationen und anderen statischen Coroutining-Mechanismen beträchtlich – vor der Unifikation liegen die zu prüfenden Argumente nicht vor und nachher ist es zu spät. Am einfachsten ist wohl die Implementation von MU-Prologs Wait-Deklarationen, da es dabei nicht auf die Argumente, sondern auf das Ergebnis der Unifikation ankommt.

5.2.5 Statische Methoden

Es wurde oft versucht, Programme mit Steuerungsanweisungen in Prolog zu übersetzen, um die Laufzeitnachteile der dynamischen Methoden zu eliminieren.

Ein Beispiel ist die Methode von Bruynooghe, de Schreye und Krekels [BdSK86], die Programme mit einer beliebigen, auf Instantiierungsmustern beruhenden Berechnungsregel in normale Prolog-Programme umformt, die sich genauso verhalten.

Ausgangspunkt ist ein symbolischer Zielaufruf mit einem Instantiierungsmuster anstatt der aktuellen Parameter – die Berechnungsregel beruht ja auf solchen Mustern. Aus diesem Zielaufruf wird nun durch abstrakte Interpretation des Programms nach der gegebenen Berechnungsregel eine Abart des Beweisbaums gewonnen, ein symbolischer Ableitungsbaum (symbolic trace tree). Der Unterschied zum Beweisbaum liegt wieder in der Betonung der Instantiierungsmuster. Aus diesem Baum kann auf einfache Weise ein Prolog-Programm gewonnen werden, das unter der normalen Prolog-Berechnungsregel den gleichen symbolischen Ableitungsbaum hat, also äquivalent ist.

Der Haken an der Sache ist, daß solche Bäume im allgemeinen unendlich sind, und die daraus abgeleiteten Prolog-Programme dementsprechend auch. Allerdings sind sich Teilbäume oft so ähnlich, daß sie mit ein wenig Arbeit zusammengelegt werden können. Dadurch wird der unendliche Baum zum endlichen Graphen und auch das zugehörige Prolog-Programm ist endlich groß.

Neben einigen händisch transformierten Beispielprogrammen wurde die Methode in einem halbautomatischen System eingesetzt.

Der praktischen Verwendung stehen einige Probleme entgegen:

- Das Resultat spiegelt die Berechnungsregel nur dann wieder, wenn der Aufruf dem im symbolischen Ziel vorgegebenen Instantiierungsmuster folgt. Dadurch verliert man einen der Hauptvorteile von Steuerungs-Mechanismen, die Invertierbarkeit. Als Ausweg bietet sich an, ein Prolog-Programm für jedes mögliche Instantiierungsmuster zu erzeugen, was einerseits Platz kosten würde, andererseits nur für einfache Muster funktioniert – z.B. dürfte Warten auf Grundinstantiierung Probleme bereiten.

- Der Geruch der Unentscheidbarkeit haftet der abstrakten Ausführung mit Abbruch bei Ähnlichkeit von Teilbäumen an, sodaß eine Automatisierung der Methode schwierig sein dürfte.

- Diese Klippe kann wahrscheinlich umschifft werden, wenn in Kauf genommen wird, daß die Instantiierungsinformation nicht immer exakt ist. Das hat zur Folge, daß entweder die Berechnungsregel nicht exakt befolgt wird (z.B. werden bei konservativem Verhalten die Ziele später aufgeweckt), oder daß doch auf Steuerung zur Laufzeit zurückgegriffen werden muß.

Es wäre auch interessant zu wissen, wieviel Zeit diese Methode einspart, insbesondere bei Programmen realistischer Größe.

Seki und Furukawa [SF87] verwenden verschiedene Spielarten der Entfaltungs/Faltungs-Methode (unfold/fold, [Dar85]), um Generate&Test-Program-

me zu transformieren. Interessant ist vor allem die dritte Methode (Unfold/fold with promotion), mit der es gelingt, die Grenzen des Coroutining zu überwinden und z.B. Permutation Sort in Sortieren durch Auswahl, einen Algorithmus mit quadratischer Komplexität, umzuwandeln. Da diese Methode beträchtliche Kreativität erfordert, liegt ihre Automatisierung wohl auch noch in weiter Ferne.

Statische Methoden der Steuerung dürften also das dynamische Coroutining wohl nicht ersetzen, insbesondere aufgrund der mangelnden Automatisierung, sind aber in Fällen, in denen Coroutining nicht die gewünschten Resultate liefert, sicherlich eine gute Ergänzung des Repertoires.

5.3 Konsistenztechniken

5.3.1 Metainterpreter

Metainterpreter erlauben zwar die Implementation von CLP-Sprachen, erleichtern sie aber nicht im gleichen Maß wie bei vielen anderen Sprachen. Wie beim Coroutining muß die Unifikation erweitert werden, was die gleichen Nachteile zur Folge hat: Ein relativ komplizierter Metainterpreter und langsame Ausführung. Nichtsdestoweniger sind viele experimentelle CLP-Systeme als Metainterpreter implementiert worden, z.B. ProBoole, CLP(Σ^*) [Wal89], CAL, Cosylog [BP89], CLP*(X) und CS-Prolog [KOM87].

Lim und Stuckey beschreiben die Implementation eines Metainterpreters für ein Sprache, die Konsistenztechniken unterstützt [LS90]. Die Hauptfrage ist dabei die Repräsentation der Bereichsvariablen. Sie werden durch eine Liste von Paaren der Form (Variable, Wertebereich) dargestellt. Die Bereiche selbst werden durch Bitfelder, also durch ganze Zahlen repräsentiert. Am schwierigsten ist neben der Implementation der Unifikation ihre Erweiterung auf Fälle mit Bereichsvariablen oder Ungleichheit. Das Ergebnis ist von der Geschwindigkeit her unbrauchbar.

Um die Unifikation nicht implementieren zu müssen, können Neumerkels Metastrukturen [Neu90] verwendet werden. Holzbaur implementiert CLP($\Re$) und endliche Bereiche mit Forward Checking mittels Metastrukturen [Hol90].

Bei der Implementation von Forward Checking verwendet Holzbaur drei Sorten von Metastrukturen:

Bereichsvariablen. Der Zugriff auf den Bereich erfolgt über Zugriffsprädikate. Dies erlaubt es, verschiedene Mengenimplementationen zu verwenden.

Verantwortliche Variablen, ähnlich wie in Abschnitt 5.2.1 dargestellt. Einige Unterschiede ergeben sich daraus, daß die Ziele bei der Instantiierung nicht sofort aufgeweckt werden können, sondern überprüft werden müssen und eventuell weiterschlafen.

Variablen, die ungleich anderen Variablen sind. Diese Sorte dient nicht nur als Spezialisierung der verantwortlichen Variablen, sondern ermöglicht auch das sofortige Versagen von $X \neq X$.

Dazu kommen alle Kombinationen dieser Sorten. Für jede Kombination aus zwei Metastrukturen muß eine `meta_meta_unify`-Klausel geschrieben werden – in diesem Fall also 28 Klauseln. Als Ausweg bietet sich an, nur die allgemeinste Kombination zu verwenden und eventuell nicht gebrauchte Teile mit einem neutralen Element zu belegen. Die gleichen Probleme müssen übrigens auch bei einer maschinennahen Implementation beachtet werden.

5.3.2 Interpreter, WAM und VAM

Bisher wurden CLP-Sprachen immer auf Prolog-Interpretern aufbauend implementiert, z.B. CLP($\Re$), CHIP und Prolog III. Die Behandlung von neuen Wertebereichen und Constraints läuft dabei über eine vom zugrundeliegenden Implementationsprinzip relativ unabhängige Schnittstelle.

Bei der Implementation auf einem Compiler könnten natürlich Constraint-Aufrufe vereinfacht und spezialisiert werden. Dazu wäre allerdings ein relativ großer Analyseaufwand notwendig. Es ist unklar und wahrscheinlich vom Berechnungsbereich abhängig, wie vorteilhaft sich solche Optimierungen auswirken. Im Sinne der Einfachkeit wurde bei ARISTO bisher darauf verzichtet.

Der Vorteil eines Compilers liegt einerseits in der erhöhten Effizienz beim Resolutions-Teil eines Programms. Andererseits verringert ein Compiler den Abstand zwischen vordefinierten, maschinennah geschriebenen Constraints und den vom Benutzer mittels Logik-Programmierung definierten Constraints. Benutzerdefinierte Constraints werden dadurch eventuell erst benutzbar, die Sprache gewinnt an Ausdruckskraft.

Die Constraint Logic Programming Shell

Um die experimentelle Implementation von CLP-Sprachen zu ermöglichen, ohne die hohen Laufzeiten bei Metainterpretern oder die hohen Entwicklungszeiten bei integrierten Systemen in Kauf nehmen zu müssen, entwickelten Lim und Stuckey die Constraint Logic Programming Shell (CLPS) [LS90]. Grundlage ist ein Prolog auf Basis der WAM. Dieses wurde um den Datentyp Löser-Variable erweitert. Löser-Variable sind Metastrukturen recht ähnlich, rufen aber bei der Unifikation nicht ein Prolog-Prädikat, sondern über eine C-Schnittstelle einen Constraint-Löser auf.

CLPS stellt Schnittstellen für drei Arten von Lösern zur Verfügung, um die Verwendung verschiedener Arten von schon existierenden oder eigens gebauten Lösern zu ermöglichen:

Black-Box-Löser nehmen ein System von Constraints und beantworten die Frage nach der Erfüllbarkeit, ob also die Constraints vereinbar sind oder nicht. Bei der normalen Unifikation entspricht das der Frage, ob alle Unifikationen erfolgreich sind.

Diese Schnittstelle ist vor allem für schon existierende Constraint-Löser gedacht, z.B. Mathematik-Pakete wie MAPLE.

CLPS merkt sich alle aufgetretenen Constraints und sendet sie alle jedesmal dem Constraint-Löser, wenn sich etwas ändert, also wenn ein Constraint in der Ableitung vorkommt oder durch Unifikation ein Gleichheitsconstraint hinzukommt. Das Backtracking wird von CLPS verwaltet.

Da es sehr teuer sein kann, den Constraint-Löser jedesmal aufzurufen, wenn ein Constraint dazukommt, stellt CLPS Prädikate zur Verfügung, um Aufrufe des Lösers zu steuern.

Vereinfachende Löser nehmen ein System von Constraints, beantworten die Frage nach der Erfüllbarkeit und liefern ein gleichwertiges, vereinfachtes System von Constraints zurück, das dann das Gesendete ersetzt. Beim Beispiel der Unifikation wird die Termgleichung durch den allgemeinsten Unifizierer (mgu) ersetzt, z.B. wird $\{f(a,X) = f(Y,b)\}$ durch $\{X = b, Y = a\}$ ersetzt. Der Löser kann sich so beim nächsten Durchgang eine Menge Arbeit ersparen. Solche Löser sind zwar inkrementeller als Black-Box-Löser, für das Backtracking sorgt aber immer noch CLPS.

Komplexe Löser nehmen ein Constraint und fügen ihn der Menge der Constraints hinzu, die der Löser schon speichert, und überprüft die Erfüllbarkeit. Ein komplexer Löser muß die Fähigkeit zum Backtracking besitzen.

Solche Löser werden wohl speziell für CLP-Sprachen gebaut werden, da inkrementelle, rücksetzende Constraint-Löser selten sind.

Die Entwicklung eines komplexen Lösers für endliche Bereiche benötigte eine ähnlich kurze Zeit wie die eines Metainterpreters (6-8 Personenstunden), die Laufzeiten waren aber um Größenordnungen kleiner. Sie reichen zwar nicht an CHIP heran, für die experimentelle Verwendung sollten sie aber genügen.

Als Nachteil ist zu vermerken, daß CLPS keine Definition von Constraints in Prolog erlaubt.

Integrierte Systeme

Dieser Abschnitt beschreibt die Implementation von endlichen Bereichen und Konsistenztechniken in CHIP [VH89] und ARISTO.

Als Grundlage für CHIP diente MU-Prolog, ein structure-sharing Interpreter, der schon einen Coroutining-Mechanismus enthielt und sich daher gut eignete. Die Erweiterungen wirken sich nur in geringem Maße auf die Effizienz aus, wenn sie nicht gebraucht werden.

Beim Structure-Sharing wird eine Variable von zwei Zeigern dargestellt; Diese werden nicht benötigt, solange die Variable frei ist. Bei einer Bereichsvariablen zeigt ein Zeiger auf den Wertebereich, der andere auf die Liste der schlafenden Ziele. Die Bereiche selbst werden auf dem globalen Stapel angelegt und enthalten ein Bitfeld mit einem Bit für jedes potentielle Element, eine Abbildung von den Bits auf die Elemente, und die Anzahl der Elemente. Wenn sich eine Bereichsvariable ändert, wird die Änderung direkt an der Variablen bzw. am Bereich vorgenommen und der alte Wert wird auf dem Trail gestapelt (Werte-Trail).

Der Unifikationsalgorithmus muß um folgende Fälle erweitert werden:

Unifikation einer Konstante mit einer Bereichsvariablen. Wie die Unifikation mit einer normalen Variablen, versagt aber, wenn die Konstante nicht im Bereich enthalten ist.

Unifikation einer einfachen und einer Bereichsvariablen. Hier wird die einfache Variable an die Bereichsvariable gebunden. Damit das immer funktioniert, müssen Bereichsvariablen immer global sein.

Unifikation zweier Bereichsvariablen. Ist der Durchschnitt der beiden Bereiche leer, versagt die Unifikation. Wenn er ein Element enthält, werden beide Variablen damit instantiiert. Ansonsten wird der Bereich einer Variablen auf den Durchschnitt gesetzt und die andere Variable an diese gebunden.

Bei Veränderungen von Bereichsvariablen werden auch die schlafenden Ziele aufgeweckt.

Forward-deklarierte Prädikate warten zunächst, bis nur noch eine Bereichsvariable übrig ist. Dann wird das Prädikat mittels eines rekursiven Interpreteraufrufs für jedes Element des Bereichs einmal aufgerufen, und die Elemente, bei denen es versagt, aus dem Bereich gestrichen. Lookahead-deklarierte Prädikate werden ähnlich behandelt.

Die vordefinierten Prädikate werden als Spezialisierungen der allgemeinen Mechanismen behandelt.

Da ARISTO auf der WAM und Structure Copying basiert, unterscheidet es sich in einigen Punkten von CHIP. So werden Bereichsvariablen durch eine Struktur dargestellt. Die Wertezelle der Variable enthält einen Zeiger auf diese Struktur. Ein weiterer Unterschied ergibt sich bei der Ausführung aufgeweckter Constraints. Hier ist eine ähnliche Vorgangsweise angezeigt wie beim Coroutining. Allerdings muß darauf geachtet werden, daß Constraints einer Variable mehrmals aufgeweckt werden können – beim Coroutining ist das nicht der Fall.

Kapitel 6

Anwendungen

6.1 Coroutining

6.1.1 Beispiele

Im Kapitel 3 sind Beispiele zu finden, die die Bedeutung von Coroutining für die Logik-Programmierung zeigen. Hier einige Beispiele aus der Praxis.

Beim Färben eines Graphen müssen zunächst die Daten angegeben werden. Dies erfolgt durch den Aufruf `graph(L)`, der sein Argument mit der Liste der Knoten unifiziert und die $\neq$-Constraints ausführt:

```
graph([N1, ...]) :- N1\=N2, ....
```

In ARISTO können Constraints derzeit aber nur an Bereichsvariablen gehängt werden. Andererseits kann `L` nicht mit Bereichsvariablen gefüllt werden, bevor bekannt ist, wie lange die Liste sein soll. Daraus ergibt sich, daß sowohl `graph/1` vor `in/2` aufgerufen werden muß als auch umgekehrt.

Als Ausweg würde ein Prolog-Programmierer `L` zunächst mit einer entsprechenden Liste instantiieren, dann `in/2` und schließlich `graph/1` aufrufen. Diese Lösung enthält allerdings durch das mehrmalige Vorkommen von `L` unnötige, änderungsfeindliche Redundanz, die man sich durch Coroutining ersparen kann:

```
colour(L, N) :- freeze(L, L in 1..N), graph(L), labeling(L).
```

Nachdem `L` im Kopf von `graph/1` instantiiert wurde, wird sofort das Ziel `L in 1..N` aufgerufen, sodaß die danach ausgeführten Constraints schon Bereichsvariablen vorfinden.

Planare Graphen lassen sich gut bildlich darstellen. Um beim Färben eines solchen Graphen zusehen zu können[1], müssen Flächen angezeigt werden, sobald sie eingefärbt werden. Übersetzt in ARISTO heißt das, daß eine Anzeigeroutine aufgerufen wird, sobald eine Bereichsvariable instantiiert wird. Dazu wird im Hauptprogramm ein Aufruf von `disp/1` eingefügt, der für jedes Mitglied seines Arguments einmal das Prädikat `polygon/2` aufruft:

```
:- delay polygon(ground, any).
polygon(Colour, P) :- x_fill_polygon(Colour, P).
polygon(Colour, P) :- x_fill_polygon(0, P), fail.
```

Der Aufruf von `polygon/2` wird also erst ausgeführt, wenn `Colour` mit einer Farbe instantiiert wird. Dann wird das entsprechende Vieleck in dieser Farbe gemalt. Beim Backtracking wird es wieder auf weiß zurückgesetzt.

6.1.2 Allgemeine Verwendung

Die meisten Verwendungsmöglichkeiten von Coroutining sind leicht zu verstehen und anzuwenden. Naish bespricht in [Nai86a] verschiedene Heuristiken für die Steuerung. Das prinzipielle Ziel ist es, einen möglichst kleinen Teil des Ableitungsbaums zu durchsuchen. Da die Länge von erfolgreichen Ableitungen fix und daher von der Steuerung unabhängig ist, beziehen sich die Heuristiken auf das Versagen. Naish nennt zwei Grundregeln, nämlich, versagende Teilbäume zu vermeiden, und, falls doch einer abgesucht wird, das Versagen früh zu erkennen.

Dazu müssen Tests möglichst früh angewandt werden, während Ziele, die viele Lösungen haben, so spät wie möglich gewählt werden sollten: versagt ein Test, müssen die anderen Ziele nicht mehr abgeleitet werden. Eine ähnliche Problematik stellt sich auch bei der Optimierung von Datenbankabfragen [Sla89].

Etwas schwieriger dürfte die Erzeugung von Delay-Deklarationen für rekursive Prädikate sein. Naish [Nai86a] beschreibt einen Algorithmus zur Erzeugung von Coroutining-Deklarationen, der im Präprozessor von MU-Prolog verwendet wird.

Rekursive Klauseln haben normalerweise Argumente, die mit jedem Rekursionsschritt einfacher werden, sodaß sie schließlich nicht mehr auf den Kopf

[1] Dies war außerordentlich wichtig für die Verbesserung des Programms – eine Animation sagt mehr als tausend Traces.

passen und daher den Abbruch herbeiführen. Ist beim Aufruf keines dieser Argumente instantiiert, so paßt der Klauselkopf immer auf den Aufruf und kann so zu einer Endlosschleife führen. Daher müssen die Delay-Deklarationen dafür sorgen, daß mindestens eines dieser Argumente genügend[2] instantiiert ist. In

```
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

wird z.B. das erste und das dritte Argument vereinfacht. Daher müssen die Delay-Deklarationen dafür sorgen, daß das erste oder das dritte Argument instantiiert ist.

Gibt es mehrere rekursive Klauseln im Prädikat, so werden zunächst die Delay-Deklarationen der einzelnen Klauseln erstellt. Die endgültigen Delay-Deklarationen müssen so gewählt werden, daß sie erst erfüllt sind, wenn von jeder Klausel mindestens eine Delay-Deklaration erfüllt ist. Dazu nimmt man jeweils eine Deklaration aus jeder Klausel und bildet den Durchschnitt dieser Deklarationen, d.h. von jeder Argumentspezifikation die strengste. Aus allen Kombinationen bilden sich so die Delay-Deklarationen des Prädikats, wobei im allgemeinen einige redundant sind.

Das Prädikat `merge/3` entspricht der Mischoperation beim Sortieren durch Mischen.

```
merge([], L, L).
merge(L, [], L).
merge([N|X], [M|Y], [N|Z]):- N<M, merge(X, [M|Y], Z).
merge([N|X], [M|Y], [M|Z]):- N>=M, merge([N|X], Y, Z).
```

Die Delay-Deklarationen für die dritte und vierte Klausel lauten

```
:- delay merge(nonvar, any, any).
:- delay merge(any, any, nonvar).
```

bzw.

```
:- delay merge(any, nonvar, any).
:- delay merge(any, any, nonvar).
```

Die Kombination ergibt:

[2] d.h. genauer als im rekursiven Aufruf

```
:- delay nrev(nonvar, any).
nrev([],[]).
nrev([X|Rest],Ans) :- nrev(Rest,L), append(L,[X],Ans).

:- delay append(nonvar, any, any).
:- delay append(any, any, nonvar).
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Abbildung 6.1: Naïve Reverse mit Delay-Deklarationen

```
:- delay merge(nonvar, nonvar, any).
:- delay merge(nonvar, any, nonvar).
:- delay merge(any, nonvar, nonvar).
:- delay merge(any, any, nonvar).
```

Die zweite und dritte Deklaration sind überflüssig und können weggelassen werden, da die vierte in jedem Fall früher erfüllt ist.

6.1.3 Effizienz

Effizienz ist eine im Computerbereich häufig verwendete und auch vielgeschmähte Ausrede, um Neuerungen nicht implementieren oder verwenden zu müssen. Um hier nicht in die übliche Polemik zu verfallen, werde ich versuchen, Coroutining anhand von Sachargumenten (Zahlen) zu bewerten. Dazu ist zu bemerken, daß WAM-Emulator und Coroutining-Mechanismus auf Einfachheit und Erweiterbarkeit ausgelegt sind und nicht optimiert wurden.

Am einfachsten ist der Fall zu messen, in dem Delay-Deklarationen angegeben wurden, weil es guter Programmierstil ist, beim Aufruf das Ziel aber sofort, also ohne Verschiebung ausgeführt werden kann. Hierzu wurde der beliebte „Naïve Reverse“-Benchmark mit Delay-Deklarationen versehen (Abbildung 6.1).

Dieses Programm zählt sicher zu den schlimmsten Fällen, da das in diesem Programm fast ausschließlich verwendete `append/3` sehr einfach und in jeder Hinsicht kurz ist, sodaß der Coroutining-Overhead besonders stark zum Tragen kommt. Als Ergebnis sank die Geschwindigkeit von 71 KLIPS[3] auf 41 KLIPS,

[3]LIPS = logische Inferenzen pro Sekunde, ein gebräuchliches Maß für die Geschwindigkeit

also um 43%. Dieser Overhead kann natürlich leicht eliminiert werden, indem die überflüssigen Delay-Deklarationen auskommentiert werden.

Interessanter ist der Fall, in dem die Delay-Deklarationen wirklich gebraucht werden. Der Vergleich fällt hier schwerer, da ein äquivalentes Prolog-Programm erstellt werden muß, was im allgemeinen nicht so einfach ist. Beim N-Damen-Problem ist der Aufwand allerdings relativ gering. Die Programme sind im Abschnitt 6.2.3 abgebildet. Auch hier fällt das Coroutining relativ stark ins Gewicht, sodaß man sich ein Bild machen kann, wieviel Coroutining maximal kostet. Die Ergebnisse sind in den Abbildungen 6.8 und 6.9 zu finden. Coroutining kostet in diesem Fall also 17% bis 50% der Geschwindigkeit. Dies muß abgewogen werden einerseits gegen den durch Coroutining eingesparten Entwicklungs- und Wartungsaufwand, andererseits gegen den Zeitaufwand für die Beschleunigung eines Programms um den Faktor 1,2–2.

6.2 Konsistenztechniken

6.2.1 Allgemeine Verwendung

Van Hentenryck gibt in [VH89, Abschnitt 5.2] Hinweise zur Lösung von CSPs. Die generelle Vorgangsweise besteht darin, zunächst die Constraints zu erzeugen und dann eine Suche einzuleiten.

Zu den Constraints ist noch zu bemerken, daß redundante Constraints dem System manchmal entscheidend bei der Suche helfen können (siehe Abschnitt 6.2.5).

Richtig interessant wird es bei den Entscheidungen, wo sich folgende Fragen stellen:

Wie wird entschieden? Wird eine Variable instantiiert (`indomain/1`) oder werden neue, nicht-redundante Constraints hinzugefügt?

Was wird zuerst entschieden? Welche Variablen sind von der Entscheidung direkt betroffen?

Wofür fällt die Entscheidung zuerst? Mit welchem Wert wird instantiiert bzw. welcher von mehreren alternativen Zusatz-Constraints wird zuerst ausprobiert. Diese Frage spielt nur bei der Suche nach einer oder der optimalen Lösung eine Rolle, dort ist sie aber unter Umständen sehr wichtig.

---

von Prolog-Systemen

Messergebnisse für	Computer	MI/s
ARISTO	DG Aviion 5000	20
	DG Aviion 200	16.7
CHIP [VH89]	DEC VAX 11/785	1
CLPS [LS90]	Sun 3/140	3
Meta-Prolog [Hol90]	Apollo DN 3000	3

Tabelle 6.1: MIPS-Annahmen für verschiedene Systeme

Hier spielen Heuristiken eine große Rolle. Gute Heuristiken sind aber meist nicht offensichtlich. Da der Aufwand zum Einbau einer Heuristik relativ klein ist, können geeignete Heuristiken auch experimentell gefunden werden. Nudel stellt in [Nud83] ein statistisches Modell auf, das er zur Erstellung von Heuristiken verwendet.

6.2.2 Zu den Messergebnissen

Gemessen wurde User-Zeit auf einem Data General Aviion 5000 Server mit einem Motorola 88100-Prozessor mit 20 Mhz und 16 Mb RAM bzw. einer Aviion 200 Workstation mit 16.7 Mhz und 8 Mb unter DG/UX 4.30. Die speicherintensiven Messungen wurden auf dem Server vorgenommen. Um einen fairen Vergleich mit anderen Systemen zu ermöglichen, werden nicht die Zeiten, sondern die Anzahl der ausgeführten Instruktionen verglichen. Tabelle 6.1 zeigt, wieviele MIPS für die verschiedenen Maschinen angenommen wurde.

6.2.3 Das N-Damen-Problem

Das N-Damen-Problem wird sowohl in der Literatur über CSPs häufig als typisches CSP angeführt, ist aber auch in der Logik-Programmierung als Beispiel für Generate&Test-Programme sehr beliebt. Es könnte als Benchmark für Coroutining und vielleicht auch für die Konsistenztechniken eine ähnliche Bedeutung und ähnliche Fragwürdigkeit erlangen wie Naïve Reverse für normales Prolog. Hier wird es verwendet, um verschiedene Ansätze zu seiner Lösung und verschiedene Systeme zu vergleichen.

Das Problem besteht darin, N Damen so auf einem $N \times N$-Brett zu plazieren, daß keine Dame eine andere schlagen kann.

Die Grundlage aller Lösungen in der Logik-Programmierung bildet das

```
queens(N,L):-
        integers(N,L1),
        permutation(L1,L),
        safe(L).

integers(0,[]).
integers(N,[N|T]):- N>0, plus(N1,1,N), integers(N1,T).

permutation(L,[H|T]):-
        delete(H,L,R),
        permutation(R,T).
permutation([],[]).

delete(X,[X|Xs],Xs).
delete(X,[Y|Ys],[Y|Rs]):- delete(X,Ys,Rs).

safe([]).
safe([X|Xs]):-
        noattack(X,Xs),
        safe(Xs).

noattack(X,L):- noattack(X,L,1).

noattack(X,[],Nb).
noattack(X,[Y|Ys],Nb):-
        notsum(X,Nb,Y),
        notsum(Y,Nb,X),
        Nb1 is Nb+1,
        noattack(X,Ys,Nb1).

notsum(X,Y,Z):- Z is X+Y,!,fail.
notsum(_,_,_).
```

Abbildung 6.2: Generate&Test-Programm für das N-Damen-Problem

```
queens(N,L):-
        integers(N,L1),
        safe(L),
        permutation(L1,L).

:- delay safe(nonvar).
:- delay noattack(any,nonvar,any).
:- delay notsum(nonvar,nonvar,nonvar).
```

Abbildung 6.3: Standard-Backtracking mittels Coroutining – Unterschiede zum Generate&Test-Programm

```
queens(N,L):-
        integers(N,L1),
        queens(L,[],L1).

queens([], Placed, []).
queens([X|Xs], Placed, Values) :-
        delete(X,Values,NewValues),
        noattack(X,Placed),
        queens(Xs,[X|Placed],NewValues).
```

Abbildung 6.4: Standard-Backtracking ohne Coroutining – Unterschiede zu Generate&Test

Generate&Test-Programm in der Abbildung 6.2. `permutations/2` ist der Generator, `safe/1` der Tester. Aus diesem Programm kann durch Hinzufügen der Delay-Deklarationen und durch Aufruf der Test-Phase vor der Generate-Phase leicht ein Standard-Backtracking-Verhalten gewonnen werden (Abbildung 6.3). Bei diesem Problem ist es auch nicht schwierig, ein Standard-Backtracking-Programm ohne Verwendung von Coroutining zu erhalten (Abbildung 6.4): `permutations/2` und `safe/1` werden verschmolzen und ergeben das Prädikat `queens/3`. Das Resultat ist zwar kürzer als das Generate&Test-Programm, aber schwerer zu schreiben und zu verstehen.

Auch das Forward-Checking-Programm wird aus dem Generate&Test-Programm abgeleitet (Abbildung 6.5). Allerdings ist es einfacher und schneller,

```
queens(N,L):-
        length(L,N),
        L in 1..N,
        alldifferent(L),
        safe(L),
        labeling(L).

length([],0).
length([X|Y],N):- N>0, plus(N1,1,N), length(Y,N1).

labeling([]).
labeling([X|Y]):- indomain(X), labeling(Y).

notsum(X,Y,Z):- notplus(X,Y,Z).
```

`alldifferent/1` und `notplus/3` sind vordefinierte Constraints.

Abbildung 6.5: Forward Checking – Unterschiede zu Generate&Test

explizit anzugeben, daß alle Variablen verschiedene Werte annehmen müssen (`alldifferent/1`) und die Wertzuweisung schließlich mit `labeling/1` vorzunehmen anstatt beides mit `permutation/1` auszudrücken. `alldifferent/1` kann nämlich aktiv verwendet werden, während `permutations/2` eher ein Standard-Backtracking-Verhalten zeigt. `labeling/1` ist ein einfacher Generator. Er nimmt eine Liste von Variablen und instantiiert sie mittels `indomain/1`. Diese Vorgangsweise liefert auch die Grundlage für die Anwendung bestimmter Heuristiken.

```
labeling([]).
labeling([F|T]):-
        deleteff(X,[F|T],Ls,[size]),
        indomain(X),
        labeling(Ls).
```

Abbildung 6.6: Forward Checking mit Auswahl der kleinsten Variablen – Unterschied zu Forward Checking

N	Lösungen
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184
16	14772512

Tabelle 6.2: Anzahl der Lösungen des N-Damen-Problems

Haralicks und Elliots [HE80] First-Fail-Heuristik, die Variable mit dem kleinsten Wertebereich zu Instantiierung zu wählen, läßt sich durch eine einfache Änderung in `labeling/1` realisieren (Abbildung 6.6). Statt der erstbesten Variablen wird mit dem vordefinierten Prädikat `deleteff/4` die Variable mit dem kleinsten Wertebereich ausgewählt.

Das letzte gemessene Programm versucht die Damen möglichst zentral zu plazieren. Unter den Variablen mit gleicher Bereichsgröße wird die der Mitte nächste ausgewählt, bei der Instantiierung werden die Werte von der Mitte zum Rand hin durchprobiert. Abbildung 6.7 zeigt eine einfache Version, die gemessene Version ist durch speichersparende Techniken etwas unansehnlich, läuft aber wegen geringeren Pagings realwesentlich schneller und kann daher für größere Probleme verwendet werden.

Es wurden zwei Problemstellungen gemessen: Eine Lösung zu finden und alle Lösungen zu finden. Wegen der explosionsartig steigenden Zahl von Lösungen (siehe Tabelle 6.2) ist es nur für kleine N möglich, alle Lösungen zu finden. Die Vorteile heuristischer Vorgangsweisen, die erst bei größerem N ihre Wirkung entfalten, kommen daher nur beim Suchen nach einer Lösung zum Vorschein. Andererseits gleicht die Suche nach allen Lösungen die Zufälle bei der Suche nach einer Lösung aus.

In Abbildung 6.8 sind die Messergebnisse für das Finden aller Lösungen dargestellt (Meßwerte unterhalb der Messgenauigkeit werden nicht gezeigt).

```
queens(N, L):-
        length(L, N),
        L in 1 .. N,
        safe(L),
        reorder(L,L1),
        qlabeling(L1, N).

qlabeling(L,N):- integers(N,L1), reorder(L1,L2), labeling(L,L2).

labeling([],_).
labeling([F|T],L):-
        deleteff(X,[F|T],R, [size]),
        member(X,L),
        labeling(R,L).

%reorder(A,B) orders the middle elements of A as the first of B
%e.g. reorder([1,2,3,4,5],[3,2,4,1,5])
reorder(L1,L2):-
        length(L1,N),
        N2 is N/2,
        length(F,N2),
        append(F, R, L1),
        rev(F,RF),
        interleave(R,RF,L2).

interleave([],[],[]).
interleave([],[H|T],[H|T]).
interleave([H|T],[],[H|T]).
interleave([X|T1],[Y|T2],[X,Y|T3]):- interleave(T1,T2,T3).
```

Abbildung 6.7: Forward Checking, letzte Version – Unterschied zu Forward Checking

$\star$ ... Generate&Test
$\circ$ ... Standard-Backtracking über Coroutining
$\bullet$ ... Standard-Backtracking in Prolog
$\triangleright$ ... Forward Checking
$\triangleleft$ ... Forward Checking mit Auswahl des kleinsten Bereichs
$\diamond$ ... CLPS mit Forward Checking

Abbildung 6.8: N-Damen-Problem – alle Lösungen

Standard-Backtracking ist wesentlich schneller als Generate&Test. Dabei unterscheidet sich die Coroutining-Version von der in normalem Prolog geschriebenen um den konstanten Faktor 1,7; nur am Anfang ist der Unterschied etwas größer. Forward Checking ist noch um einiges schneller, der Unterschied ist etwa proportional zu N. Es bleibt nicht mehr viel Spielraum für aufregende Leistungen, sodaß die First-Fail-Heuristik hier keine besonderen Verbesserungen ergibt. Bei kleinem N wirkt sich der größere Aufwand nachteilig aus, dafür bleiben ab $N = 10$ die Kosten pro Lösung annähernd konstant.

In Abbildung 6.9 sind die Ergebnisse für die Suche nach einer Lösung zu sehen. Im Unterschied zur Suche nach allen Lösungen verkleinert sich der Unterschied zwischen den Coroutining-Varianten mit steigendem N. Bei $N = 28$ ist der Faktor auf 1,2 gefallen. Der Unterschied zwischen Standard-Backtracking und Forward Checking ist auch hier direkt proportional zu N. Doch auch bei Forward Checking steigt die Suchzeit exponentiell. Im Vergleich dazu verhält sich Forward Checking mit Auswahl der kleinsten Variable sensationell.

Abbildung 6.10 stellt schließlich die Ergebnisse von Forward Checking mit verschiedenen Heuristiken bei größeren, geraden N dar. Dabei tritt die nichtdeterministisch quadratische Natur des Algorithmus zutage: eine quadratische Anzahl von Constraints wird je einmal aufgerufen. Teilweise kommt durch die Suche einiges an Aufwand hinzu. Im schlimmsten Fall trifft das Programm früh eine Fehlentscheidung und sucht dann lange aussichtslos herum. Die Suchzeit wurde mit fünf Minuten begrenzt. Die einfache Heuristik überschreitet diese Grenze das erste mal bei $N = 86$ und dann relativ oft.

Wesentlich besser verhält sich die Heuristik, die Dame im Zweifelsfall möglichst zentral zu setzen. Dem Wesen einer Heuristik entsprechend, verirrt sie sich auch manchmal, aber doch wesentlich seltener. Sie überschreitet die Zeitgrenze in 19 Fällen, davon fünfzehn mal bei $N < 300$, das erste Mal bei $N = 106$.

Mit der Heuristik ändern sich auch die Stellen des Verirrens. So kann durch Kombination mehrerer Heuristiken die Wahrscheinlichkeit beträchtlich gesteigert werden, in erträglicher Zeit für ein gegebenes N eine Lösung zu finden. Eine leichte andere Heuristik zeigte zwar ein ähnliches Verhalten – diesmal wurde die Grenze 21 mal überschritten – doch nur in einem Fall brauchen beide Heuristiken länger als fünf Minuten.

Zum N-Damen-Problem liegen auch Zahlen von CHIP [VH89] und CLPS [LS90] vor. CHIP führt durchwegs 1,5 mal mehr Instruktionen aus als ARISTO. Dies mag vielleicht etwas verwundern, da CHIP auf einem Interpreter basiert und ARISTO auf einem Compiler, was einen größeren Unterschied erwarten läßt. Jedoch wird nur ca. 10% der Zeit im WAM-Emulator verbracht,

$\star$ ... Generate&Test
$\circ$ ... Standard-Backtracking über Coroutining
$\bullet$ ... Standard-Backtracking in Prolog
$\triangleright$ ... Forward Checking
$\triangleleft$ ... Forward Checking mit Auswahl des kleinsten Bereichs
$\diamond$ ... CLPS mit Forward Checking

Abbildung 6.9: N-Damen-Problem – erste Lösung

Heuristiken zur Auswahl der nächsten Dame:
$\times$ … Variable mit dem kleinsten Bereich
$+$ … Variable mit dem kleinsten Bereich, im Zweifelsfall möglichst zentral
Die Fälle, in denen die Zeitschranke (6000 MI) überschritten wurde, sind auf der 10000- bzw. 12000-MI-Linie zu sehen.

Abbildung 6.10: N-Damen-Problem mit großen N

der Rest entfällt auf die Ausführung und Verwaltung der Constraints, die in beiden Systemen ähnlich implementiert und daher ähnlich effizient ist.

Im großen Anteil der Constraint-Bearbeitung liegt auch der Grund für das schlechte Abschneiden von CLPS mit einem komplexen Löser für Bereichsvariable und Konsistenztechniken. CLPS benötigt etwa 40 mal mehr Instruktionen als ARISTO. Das Interface zwischen WAM-Emulator und Constraint-Löser dürfte für Konsistenztechniken nicht optimal sein.

6.2.4 Graphenfärbung

Die Knoten eines Graphen müssen mit möglichst wenig Farben so gefärbt werden, daß jede Kante nur verschiedenfarbige Knoten verbindet. Eine bekannte Unterklasse dieses Problems ist das Vier-Farben-Problem, das sich durch Beschränkung auf planare Graphen ergibt.

Graphenfärben wurde schon intensiv untersucht, da es viele Anwendungen hat. Einige Beispiele sind Registerzuteilung im Compilerbau und Stundenpläne, Warenlagerung und Maschinenbelegungspläne im Operations Research.

Konsistenztechniken in der Logik-Programmierung erlauben nicht nur die effiziente Lösung von Graphenfärbungsproblemen, sondern ermöglichen auch die anwendungsspezifische Erweiterung und die Ausnutzung spezifischer Eigenschaften des Problems, die sonst oft Schwierigkeiten bereitet. So gibt es bei der Registerzuteilung oft zusätzliche Einschränkungen, z.B. benötigen oft gewisse Instruktionen ein Paar nebeneinanderliegender Register. Solche Constraints können leicht in das Logik-Programm eingebaut werden.

Graphenfärben ist ein Optimierungsproblem. Van Hentenryck verwendet daher `minimize_maximum/2` zu seiner Lösung, von der sich der hier gezeigte Ansatz sich in einem Punkt unterscheidet: statt Branch-and-Bound wird ein Iterative-Deepening-Verfahren verwendet. Die Anzahl der Farben wird von 1 ausgehend solange erhöht, bis eine Lösung gefunden wird.

Wie verhält sich dieser Ansatz im Vergleich zu Van Hentenrycks? Beide Programme suchen die erste Lösung für die chromatische Zahl χ und versuchen, den Graphen mit $\chi - 1$ Farben zu färben. Van Hentenryck erspart dam Computer das Durchsuchen für $1 \ldots \chi - 2$, muß dafür aber für $\chi + 1 \ldots M$ die erste Lösung suchen. Welches Verfahren günstiger ist, hängt davon ab, ob das Programm besser im Finden einer Lösung ist, oder im Beweisen, daß keine Lösung existiert.

Der Ausgangspunkt der Lösung ist das Prädikat `color/2`, das einen Graphen mit `N` Farben einfärbt.

```
color(L, N):- freeze(L, L in 1..N), graph(L), labeling(L1).
```

`graph/1` stellt dabei die Daten zur Verfügung: Es unifiziert sein Argument mit der Liste der Knoten und wendet die Constraints auf die Knoten an.

`color/2` wird nun mit aufsteigendem `N` aufgerufen, bis es eine Lösung liefert:

```
mincolor(L, N, M):- between(N, 1, M), color(L, N).
```

`M` ist eine pessimistische (zu hohe) Schätzung der minimalen Anzahl der Farben. Dabei wird angenommen, daß `indomain/1` sein Argument zuerst mit den niedrigen Werten instantiiert.

Wie bereits beim N-Damen-Problem zu sehen war, ist die Reihenfolge, in der die Variablen instantiiert werden, außerordentlich wichtig für die effiziente Lösung des Problems. Die Variable mit dem kleinsten Bereich zu wählen, genügt hier nicht. Daher wird in unentschiedenen Fällen die Variable mit der größten Anzahl von Constraints gewählt, in der Sprache der Graphentheorie also den Knoten mit dem maximalen Grad. Die Instantiierung dieser Variable verkleinert sofort die Bereiche von vielen anderen Variablen und damit den Suchraum.

Ein weiterer wichtiger Punkt ist die Vermeidung unnötiger Suche. Jede Färbung mit N Farben hat $N!$ symmetrische Lösungen. Bei Teilproblemen treten ähnliche Symmetrien auf, die nicht durchsucht werden brauchen. Wenn ein Knoten zum Färben ansteht, und bisher i Farben verwendet wurden, braucht man für den aktuellen Knoten nur $i+1$ Farben in Betracht ziehen:

- die i bisher verwendeten Farben.
- Eine noch nicht verwendete Farbe. Da alle Färbungen mit noch nicht verwendeten Farben symmetrisch sind, muß nur eine in Betracht gezogen werden.

Diese Überlegungen sind im Prädikat `labelvar/3` realisiert, dessen erstes Argument der zu färbende Knoten ist, das zweite gibt die Zahl der bisher verwendeten Farben an, das dritte die neue Zahl der Farben.

`labeling/1` und `labelvar/3` sind in Abbildung 6.11 zu sehen.

Gemessen wurde einerseits die Färbung von drei planaren Graphen, nämlich Europa, die USA und der von Gardner in einer Aprilkolumne [Gar75] als

```
labeling(X):- labeling(X,0).

labeling([],Newup).
labeling([X|Y], Up):-
        deleteff(Var, [X|Y], Rest, [size,constraints]),
        labelvar(Var, Up, Newup),
        labeling(Rest, Newup).

labelvar(X,N,N):- between(X,1,N).
labelvar(X,N,X):- X is N+1.

between(X,L,H):- H>=L, X in L..H, indomain(X).
```

Abbildung 6.11: Wertezuweisung beim Graphenfärben

Graph	Knoten	ARISTO		
Europa '89	34	0,4 MI	5,1 MI	Meta-Prolog [Hol90]
USA	50	0,6 MI		
[Gar75]	110	2,2 MI	5,5 MI	CHIP [VH89]

Tabelle 6.3: Resultate für das Vier-Farben-Problem

30,...,70 ... verschiedene Anzahl von Knoten für ARISTO
40, *50* ... verschiedene Anzahl von Knoten für CHIP [VH89]

Abbildung 6.12: Graphenfärben – Gesamtaufwand

Gegenbeweis zum Vier-Farben-Theorem angegebene Graph mit 110 Regionen. Die Ergebnisse von ARISTO im Vergleich mit anderen Systemen finden sich in Tabelle 6.3. ALICE [Lau78] benötigt für Gardners Problem 12 Sekunden auf einer IBM 370/168, einer Maschine aus den späten siebziger Jahren.

Weiters wurden pseudo-zufällige Graphen mit konstanter Dichte erzeugt, d.h. ein Graph mit n Knoten und Dichte d enthält $\frac{n(n-1)}{2}d$ zufällig verteilte Kanten. In [KJ85] und in [VH89] wurden pro n und d drei Graphen erzeugt; hier wurden je fünf Graphen gemessen, um die Zufallsschwankungen besser auszugleichen.

Die Ergebnisse der Messungen sind in Abbildung 6.12 zu finden. Gezeigt werden Durchschnittswerte und zwar nur in Fällen, in denen keiner der Graphen beim Färben mehr als 6000 MI benötigt. Bei hohen und niedrigen Dich-

30,...,60 ... ARISTO
30,...,60 ... das jeweils beste Spezialprogramm aus [KJ85]

Abbildung 6.13: Graphenfärben – Anzahl der Backtrackings

ten ist ARISTO besser als CHIP, bei mittlerer Dichte liegt CHIP voran. Der Unterschied erklärt sich aus den unterschiedlichen Optimierungsalgorithmen.

Kubale und Jackowski haben Spezialprogramme für das Graphenfärben gemessen [KJ85]. Die dort angegebenen Zeiten wurden auf einem SC-32 Computer ($\hat{=}$ IBM 360/65) gemessen und sind daher schwer zu vergleichen. Daher wurde die Anzahl der Backtrackings verglichen, die zwar kein Maß für den Aufwand, aber immerhin eines für die Komplexität darstellt. Die Ergebnisse sind in Abbildung 6.13 zu sehen. Bei den Ergebnissen von ARISTO sind die $\chi - 1$ Backtrackings nicht angeführt, die durch das Iterative-Deepening-Verfahren hervorgerufen werden. Das jeweils beste Spezialprogramm hatte zwar weniger Backtrackings, die Leistungen des Logik-Programms liegen aber nicht weit zurück und sind für seine Kürze recht beeindruckend. Durch den Einbau besserer Heuristiken können sie sicher noch gesteigert werden.

Markierungen	MI
6	2.4
7	20
8	171
9	1460
10	11200
11	381000
12	2200000

Tabelle 6.4: Aufwand zum Finden der Golomb-Lineale

6.2.5 Golomb-Lineale

Ein Golomb-Lineal ist das kürzeste Lineal mit einer gegebenen Anzahl von Markierungen, bei dem jeder Abstand zwischen zwei Markierungen maximal einmal vorkommt [Dew86]. Anwendungen finden sich in der Codierungstheorie, der Röntgen-Strukturanalyse, der Konstruktion von Chips, und der Radioastronomie.

Die Angabe läßt sich direkt und sehr deklarativ in das in Abbildung 6.14 gezeigte Programm umsetzen. Neben den Constraints, die sich direkt aus der Angabe ergeben, sind noch einige Constraints nötig, um symmetrische Lösungen zu unterdrücken. `ordered(L)` unterdrückt Lösungen, die sich aus Permutationen der Markierungen ergeben, `FirstDist #> LastDist` ist für die Spiegelung um die Längsachse zuständig.

Um den Suchbaum stärker zu beschneiden, wurden noch redundante Constraints hinzugefügt: Der Abstand zwischen den Markierungen M_i und M_j ist mindestens so groß wie ein Golomb-Lineal mit $|j-i|+1$ Markierungen[4]. Schon bei kurzen Linealen beschleunigt sich die Suche dadurch um den Faktor 4.

Bei den Entscheidungen erwies sich die Zuweisung an die Variable mit dem kleinsten Bereich und den meisten Constraints experimentell als günstig. Zur Optimierung wird nach Linealen mit immer größerer Länge gesucht, bis eine Lösung gefunden wird. Diese Strategie ist in diesem Fall etwa genauso schnell wie ein Branch-and-Bound-Algorithmus.

Die Ergebnisse sind in Tabelle 6.4 zu sehen. Das Programm fand die Golomb-Lineale mit 3–12 Markierungen in knapp eineinhalb Tagen. Zum Vergleich sei gesagt, daß ein Spezialprogramm zum Finden der Golomb-Lineale

[4] Im gezeigten Programm werden die Längen der Golomb-Lineale direkt angegeben. Sie könnten natürlich auch vom Programm errechnet werden.

```
golomb(N, L, M):-
        L in 0..M,
        Distances in 1..M,
        length(L, N),
        first(L, 0), last(L, Max),
        diffs_of_two(L,Distances),
        alldifferent(Distances),
        ordered(L),             %constraint to reduce symmetries
        first(Distances,FirstDist), last(Distances,LastDist),
        FirstDist #> LastDist, %constraint to reduce symmetries
        red(L,Distances),       %redundant constraints
        minimize(labeling(L,[size,constraints]), Max).

diffs_of_two([],[]).
diffs_of_two([H|T],S):-
        append(S1,S2,S),
        diffs_of(T,H,S1),
        diffs_of_two(T,S2).

diffs_of([],_,[]).
diffs_of([H|T],X,[S|ST]):- X+S#=H, diffs_of(T,X,ST).

red(L,Distances):-
        length(L, N), plus(N1,2,N), length(L1,N1),
        append(L1,_,[1,3,6,11,17,25,34,44,55,72,85,106]),
        append(L1,[1],Ls),
        red(L,Distances,Ls).

red([],[],Ls).
red([_|T],S,Ls):-  append(S1,S2,S), red1(T,S1,Ls), red(T,S2,Ls).

red1([],[],_).
red1([_|T],[D|DT],[M|MT]):-  D#>=M, red1(T,DT,MT).

ordered([]).
ordered([_]).
ordered([X,Y|T]):- X#<Y, ordered([Y|T]).
```

Abbildung 6.14: Programm zum Berechnen von Golomb-Linealen mit bis zu 13 Markierungen ein Monat benötigte [Dew86].

6.2.6 Scheduling in Echtzeitsystemen

Echtzeitsysteme für harte Echtzeitanwendungen müssen ein bestimmtes Zeitverhalten garantieren. Daher werden in MARS, einem fehlertoleranten, verteilten Echtzeitsystem die Echtzeittasks nach einem sogenannten time-rigid Schedule ausgeführt [Koz89]. Das Schedule wird vor Inbetriebnahme des Systems so berechnet, daß alle Garantien erfüllt sind. Zur Laufzeit muß der Dispatcher nur noch eine Scheduling-Tabelle abarbeiten.

Das System besteht aus mehreren Komponenten (CPUs), die über einen Bus miteinander verbunden sind. Alle Tasks werden periodisch ausgeführt. Die Tasks, die maximale Abarbeitungszeit, und die Komponente, auf der ein Task ausgeführt wird, werden in der Entwurfsphase vor dem Scheduling festgelegt.

Tasks werden in Gruppen, sogenannte Transaktionen eingeteilt, die über den Austausch von Nachrichten miteinander kommunizieren. Sitzen zwei miteinander kommunizierende Tasks auf verschiedenen Komponenten, so müssen für die Nachricht Bus-Zeitscheiben reserviert werden. Die meisten Tasks können unterbrochen werden, was das Scheduling erleichtert.

Folgende Einschränkungen müssen berücksichtigt werden:

- Zwei Tasks können nicht gleichzeitig auf der gleichen Komponente ausgeführt werden.
- Zwei Nachrichten können nicht gleichzeitig auf dem Bus gesendet werden.
- Eine Nachricht mußübertragen werden, nachdem der sendende Task beendet ist und bevor der oder die Empfänger begonnen haben.
- Die Zeit vom ersten bis zum letzten Task einer Transaktion ist begrenzt (maximal response time).

Bei Scheduling-Problemen wird normalerweise jeder Task durch eine Bereichsvariable dargestellt. Der Wert der Bereichsvariable entspricht dem Startzeitpunkt des Tasks. Hinzu gesellen sich Constraints und Entscheidungen.

Scheduling in MARS unterscheidet sich von der in [VH89] beschriebenen Scheduling-Aufgabe durch die periodische Ausführung und die Unterbrechbarkeit der Tasks. Dementsprechend unterscheidet sich auch die Lösung.

Tasks, deren Periode kleiner ist als die Scheduling-Periode, werden vervielfacht. Sie treten in mehreren, jeweils eine Periode voneinander entfernten Kopien auf. Da Tasks nach jeder Zeitscheibe unterbrochen werden können, wird ein Task mit n Zeitscheiben durch n Variablen dargestellt, die die Nummer der jeweiligen Zeitscheibe angeben. Nachdem die Constraints angegeben wurden, kann ein Schedule mit einem Labeling-Prädikat berechnet werden.

Der Entwurf und die Implementation des Programms benötigen etwa drei Stunden. Das Programm benötigt zum Berechnen eines Schedules mit 38 Tasks auf sieben Komponenten 2.7s (54 Millionen Instruktionen). Die Auslastung der Komponenten liegt zwischen 25% und 81%, die Auslastung des Busses ist 47%. Ein auf IDA*, einem Best-First-Algorithmus basierendes Spezialprogramm benötigte für eine etwas leichtere Aufgabe 50s (750 Millionen Instruktionen) auf einer DECStation 3100.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die Logik-Programmierung hat viele Vorteile. Sie eignet sich unter anderem besonders gut dafür, Suchprobleme auszudrücken. Prolog, die bekannteste Logik-Programmiersprache, ist ein Kompromiss zwischen der reinen Lehre der Logik-Programmierung und den Erfordernissen einer effizienten Implementation und der allgemeinen Verwendbarkeit. Dementprechend kommen auch die Vorteile der Logik-Programmierung nur beschränkt zur Geltung.

Inzwischen gibt es in mehreren Bereichen bessere Lösungen, die auch effizient implementiert werden können:

Coroutining ermöglicht die Trennung von Logik und Steuerung. Dadurch werden auch deklarativ geschriebene Programme ausführbar. Weiters bildet Coroutining die Grundlage für eine sichere Implementation von Negation und Quantoren und für die Konsistenztechniken.

Constraint Logic Programming (CLP) erlaubt die Erweiterung des Berechnungsbereichs von uninterpretierten Termen auf beinahe beliebige Wertebereiche.

Konsistenztechniken in der Logik-Programmierung unterstützen die effiziente Lösung von Suchproblemen. Durch die Einbettung in eine allgemeine Programmiersprache kann das Programm flexibel an das Problem angepaßt werden. Andererseits können in diesem Rahmen Programme zur Lösung von Suchproblemen wesentlich leichter und schneller entwickelt werden als Spezialprogramme in imperativen Programmiersprachen.

Das Anwendungsgebiet ist breit gestreut und reicht von Anwendungen im Operations Research und in der künstlichen Intelligenz bis zur Hardware-Entwicklung.

Bei den Anwendungen zeigt sich, daß Heuristiken entscheidend sind für die Größe der Probleme, die gelöst werden können. Sie lassen sich leicht in die Programme einbauen.

Die Implementation kann entweder auf der Prolog-Ebene mit meta-sprachlichen Mitteln erfolgen oder in einer maschinennahen Sprache wie C. Die Implementation erfolgt durch eine Erweiterung der Unifikation. Daher bieten sich für die Implementation auf der Prolog-Ebene besonders die Metastrukturen an. Eine solche Implementation kann zwar die Vorteile der Logik-Programmierung für sich verbuchen, ist aber beim heutigen Stand der Übersetzung von Logik-Programmiersprachen noch keine Konkurrenz zur Implementation in einer maschinennahen Programmiersprache.

Die Implementation auf Grundlage eines Compilers unterstützt die Verwendung der Erweiterungen im Rahmen normaler Prolog-Programme und erlaubt die besonders effiziente Behandlung von logik-programmierten Constraints, sodaß diese brauchbarer werden. Maschinennahe Constraint-Implementationen lassen sich dadurch auf einen kleinen Grundbestand beschränken.

Die Erweiterungen verursachen im Emulator der Warren Abstract Machine (WAM) nur einen minimalen zeitlichen Mehraufwand für Programme, die diese Erweiterungen nicht benötigen.

7.2 Ausblick

Im Bereich des Coroutining ist vor allem eine noch effizientere Implementation durch die Integration von Indexing und der Überprüfung auf Erfüllung der Deklarationen möglich. Der Mehraufwand der (nicht benötigten) Coroutining-Deklarationen beim Naïve Reverse würde dann verschwinden. Coroutining-Deklarationen können sogar dazu verwendet werden, um das Indexing zu steuern und so an Effizienz zu gewinnen.

Sicherlich ist Coroutining nicht der Weisheit letzter Schluß. Auch andere Steuerungsmechanismen wären interessant, z.B. Saraswats. Bei der Entwicklung von Suchprogrammen störte besonders das Fehlen von deklarativen

Möglichkeiten, die Auswahl von Alternativen zu beeinflußen. Angesichts der Trägheit, mit der schon Coroutining angenommen wird, ist jedoch zu bezweifeln, daß sich weitere Steuerungsmechanismen in der Logik-Programmierung allgemein durchsetzen werden.

Beim CLP-Schema steht die Erschließung neuer Berechnungsbereiche auf dem Programm. Wichtig wäre auch die Verallgemeinerung und Vereinfachung des Vorgangs, also einerseits eine Methodologie zum Entwurf der Berechnungsbereiche und der Constraints, andererseits eine gute Unterstützung der Implementation. Ein Schritt in diese Richtung sind CLPS und die Metastrukturen. Wichtig wäre auch eine Unterstützung der Definition neuer Constraints in der Logik-Sprache selbst. Am Ende dieser Entwicklung steht hoffentlich ein einfacher Mechanismus, um die Sprache um neue Datentypen zu erweitern.

Auch bei den Konsistenztechniken wäre eine bessere Unterstützung bei der Definition von Constraints wünschenswert. Derzeit lassen sich nur Konjunktionen von Constraints ausdrücken. Disjunktionen von Constraints können nur als Entscheidung benutzt werden, andere Verknüpfungen von Constraints funktionieren überhaupt nur als Tests. Als Alternative bieten sich CHIPs `forward`- und `lookahead`-Deklarationen an, die aber durch Bearbeitung der einzelnen Wertekombinationen implementiert und daher oft zu ineffizient sind.

Heuristiken sind zur Lösung von Suchproblemen außerordentlich wichtig. Daher ist eine bessere Unterstützung von Heuristiken wünschenswert, wobei einerseits eine globalere Sicht notwendig ist, andererseits schärfere Unterscheidungen getroffen werden müssen.

Für die Hardwareentwicklung folgt, daß die Entwicklung von allgemeinen Architekturen mit besonderer Eignung für die Logik-Programmierung günstiger ist als die Schaffung einer WAM in Hardware, sonst würde der aktuelle Stand der Entwicklung in Silizium festgeschrieben. Neue Entwicklungen würden entweder nicht angenommen oder die Hardware wäre obsolet.

7.3 Danksagung

Meinen Kollegen und Betreuern Thomas Graf und Andreas Krall möchte ich für die gute Zusammenarbeit und die Betreuung danken. Dank schulde ich auch Gerhard Fohler, der mir Literatur und Problemstellungen zum Scheduling in Echtzeitsystemen zur Verfügung stellte, und Christian Holzbaur und Ulrich Neumerkel für die Literatur.

Literaturverzeichnis

[BdSK86] Maurice Bruynooghe, Danny de Schreye, und Bruno Krekels. Compiling control. In *1986 Symposium on Logic Programming*, Seiten 70–77. IEEE, 1986.

[BMMW89] Alan Borning, Michael Maher, Amy Martindale, und Molly Wilson. Constraint hierarchies and logic programming. In *Logic Programming: Proceedings of the Sixth International Conference*, Seiten 149–164, Lissabon, 1989.

[Boi86] Patrice Boizumault. A general model to implement `dif` and `freeze`. In *Logic Programming: Proceedings of the Third International Conference*, Seiten 585–592, London, 1986. Springer LNCS 225.

[Boi88] Patrice Boizumault. *Prolog: L'implantation.* Masson, Paris, 1988.

[BP89] Henri Beringer und Franck Porcher. A relevant scheme for Prolog extensions: CLP(Conceptual Theory). In *Logic Programming: Proceedings of the Sixth International Conference*, Seiten 131–148, Lissabon, 1989.

[BS87] W. Buttner und H. Simonis. Embedding boolean expressions into logic programming. *Journal of Symbolic Computation*, 4:191–205, Oktober 1987.

[Car87] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Logic Programming: Proceedings of the Fourth International Conference*, Seiten 40–58, Melbourne, 1987.

[Car90] Mats Carlsson. Re: Arrays in Prolog. Usenet News "1990 Sep 25: 124816.12993@sics.se", 1990.

[CMG82] K. L. Clark, F. G. McCabe, und S. Gregory. IC-Prolog language features. In K. L. Clark und S.-A. Tärnlund, Hrsg., *Logic Programming*, Seiten 253–266. Academic Press, London, 1982.

[Col87] Alain Colmerauer. Opening the Prolog III universe. *BYTE*, 12(9):177–182, August 1987.

[Col90] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, Juli 1990.

[Dar85] John Darlington. Program transformation. *BYTE*, 10(8):201–216, August 1985.

[Dew86] A. K. Dewdney. Computer-Kurzweil. *Spektrum der Wissenschaft*, Seiten 5–11, Februar 1986.

[DP88] R. Dechter und J. Pearl. A problem simplification approach that generates heuristics for constraint-satisfaction problems. In J. E. Hayes, D. Michie, und J. Richards, Hrsg., *Machine Intelligence 11*. Oxford University Press, 1988.

[DVHS+88] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abder Aggoun, Thomas Graf, und Françoise Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS)*, Tokyo, Dezember 1988.

[FBMB90] Bjorn N. Freeman-Benson, John Maloney, und Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, Januar 1990.

[Fre78] Eugene C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, November 1978.

[Gar75] Martin Gardner. Mathematical games. *Scientific American*, 232(4):126–133, April 1975.

[GKPvC86] Françis Giannesini, Henry Kanoui, Robert Pasero, und Michel van Caneghem. *Prolog*. International Computer Science Series. Addison-Wesley, 1986.

[GLLO85] John Gabriel, Tim Lindholm, E. L. Lusk, und R. A. Overbeek. A tutorial on the Warren Abstract Machine for computational logic. Technischer Bericht ANL-84-84, Argonne National Laboratory, 1985.

[Gra87] Thomas Graf. Extending constraint handling in logic programming to rational arithmetic. Internal report, ECRC, 1987.

[Gra89] Thomas Graf. *Raisonnement sur les contraintes en programmation en logique*. Dissertation, Université de Nice – Sophia Antipolis, 1989.

[HE80] Robert M. Haralick und Gordon L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[HJL+86] Nevin Heintze, Joxan Jaffar, Chean Shen Lim, Spiro Michaylov, Peter Stuckey, Roland Yap, und Chut Ngeow Yee. *The CLP Programmer's Manual, Version 1.0*. Department of Computer Science, Monash University, Australia, 1986.

[Hol90] Christian Holzbaur. *Implementation of Constraint Based Inference Mechanisms through Extended Unification*. Dissertation, Technische Universität Wien, 1990.

[JL87a] Joxan Jaffar und Jean-Louis Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, Seiten 111–119, München, 1987. ACM SIGPLAN.

[JL87b] Joxan Jaffar und Jean-Louis Lassez. From unification to constraints. In K. Furukawa, H. Tanaka, und T. Fujisaki, Hrsg., *Logic Programming '87*, Seiten 1–18. Springer LNCS 315, 1987.

[KJ85] Marek Kubale und Bogusław Jackowski. A general implicit enumeration algorithm for graph coloring. *Communications of the ACM*, 28(4):412–418, April 1985.

[KN90] Andreas Krall und Ulrich Neumerkel. The Vienna Abstract Machine. In *Programming Language Implementation and Logic Programming (PLILP'90)*, Seiten 121–136. Springer LNCS 456, 1990.

[KOM87] Toshio Kawamura, Hayato Ohwada, und Fumio Mizoguchi. CS-Prolog: A generalized unification based constraint solver. In K. Furukawa, H. Tanaka, und T. Fujisaki, Hrsg., *Logic Programming '87*, Seiten 19–39. Springer LNCS 315, 1987.

[Kow79] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, Juli 1979.

[Kow85] Robert Kowalski. Logic programming. *BYTE*, 10(8):161–177, August 1985.

[Koz89] Christian Koza. *Garantiertes Zeitverhalten in verteilten Echtzeitsystemen.* Dissertation, Technische Universität Wien, 1989.

[Kra88] Andreas Krall. *Analyse und Implementierung von Prolog-Systemen.* Dissertation, Technische Universität Wien, 1988.

[Lau78] Jean-Louis Lauriere. A language and a program for stating and for solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[LS90] Pierre Lim und Peter J. Stuckey. A constraint logic programming shell. In *Programming Language Implementation and Logic Programming (PLILP'90)*, Seiten 75–88. Springer LNCS 456, 1990.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[McA90] David McAllester. Truth maintenance. In *Ninth National Conference on Artificial Intelligence (AAAI)*, Seiten 1109–1116, 1990.

[Mel82] C. S. Mellish. An alternative to structure sharing in the implementation of a Prolog interpreter. In K. L. Clark und S.-A. Tärnlund, Hrsg., *Logic Programming*, Seiten 99–106. Academic Press, London, 1982.

[Nai86a] Lee Naish. *Negation and Control in Prolog.* Springer LNCS 238, 1986.

[Nai86b] Lee Naish. Negation and quantifiers in NU-Prolog. In *Logic Programming: Proceedings of the Third International Conference*, Seiten 625–634, London, 1986. Springer LNCS 225.

[Nai90] Lee Naish. E-Mail Korrespondenz, 1990.

[Neu88] Gustaf Neumann. *Metaprogrammierung und Prolog.* Internationale Computer-Bibliothek. Addison-Wesley, Bonn, 1988.

[Neu90] Ulrich Neumerkel. Extensible unification by metastructures. In *Meta-90*, Leuven, Belgien, 1990.

[Nud83] Bernard Nudel. Consistent labeling problems and their algorithms: Expected complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[Pan86] H. Pangratz. *Logische Schaltkreise.* Technische Universität Wien, Institut für Datenverarbeitung, 1986.

[Pei84] L. M. Peirera. Logic control with logic. In J. A. Campbell, Hrsg., *Implementations of Prolog*, Seiten 177–193. Ellis-Horwood, 1984.

[Pic85] Christian Pichler. Prolog-Übersetzer. Diplomarbeit, Technische Universität Wien, 1985.

[Sar87] Vijay A. Saraswat. CP as a general-purpose constraint-language. In *Sixth National Conference on Artificial Intelligence (AAAI)*, Seiten 53–58, 1987.

[SF87] Hirohisa Seki und Koichi Furukawa. Notes on transformation techniques for generate and test logic programs. In *1987 Symposion on Logic Programming*, Seiten 215–223. IEEE, 1987.

[Sla89] Wolfgang Slany. Optimierung relationaler Anfragen am Beispiel der ARTHUR Implementierung. Diplomarbeit, Technische Universität Wien, 1989.

[VC86] Michel Van Caneghem. *L'anatomie de Prolog*. InterEditions, Paris, 1986.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. The MIT Press, Cambridge, Massachusetts, 1989.

[Wal89] Clifford Walinsky. CLP(Σ^*): Constraint logic programming with regular sets. In *Logic Programming: Proceedings of the Sixth International Conference*, Seiten 181–196, Lissabon, 1989.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technischer Bericht 309, SRI International, 1983.