Skript zur Vorlesung

Implementierung logischer Programmiersprachen

SS'97

Prof. Dr. Michael Hanus

Lehr- und Forschungsgebiet Informatik II
RWTH Aachen

Vorwort

Logische Programmiersprachen, deren Hauptvertreter Prolog ist, haben in den letzten Jahren eine besonders starke Verbreitung gefunden, weil entscheidende Fortschritte in den Implementierungstechniken gemacht worden sind. Heutige Prolog-Implementierungen übersetzen logische Programme in direkt oder indirekt ausführbaren Code für herkömmliche von-Neumann-Rechnerachitekturen, so daß viele Programme oder Programmteile genauso effizient ausgeführt werden wie äquivalente Programme einer imperativen Programmiersprache (C, Modula,...). Wenn man die Kosten zur Softwareerstellung berücksichtigt (die ja durch den höheren Abstraktionsgrad bei deklarativen Sprachen geringer sind), dann sind durch die heutigen Implementierungstechniken logische Programmiersprachen für viele Anwendungsprobleme im Vergleich zu herkömmlichen imperativen Programmiersprachen konkurrenzfähig oder sogar überlegen.

In dieser Vorlesung soll der heutige Stand der Implementierungstechniken für logische Programmiersprachen dargestellt werden. Hierbei soll deutlich werden, wie man durch systematische Verbesserungen von der abstrakten operationalen Semantik zu effizienten Implementierungen gelangen kann. Im Rahmen von Vorlesungen zum Übersetzerbau werden Grundkonzepte der Sprachimplementierung behandelt. Die Kenntnis dieses Stoffgebietes ist zwar vorteilhaft, aber nicht unbedingt notwendig zum Verständnis dieser Vorlesung, da hier alle notwendigen Grundlagen eingeführt werden. Die vorliegende Vorlesung versteht sich aber als wichtige Ergänzung und Vertiefung zur Übersetzerbau-Vorlesung.

Dieses Skript ist eine überarbeitete Fassung des Skripts zu einer gleichnamigen Spezialvorlesung, die ich an der Universität Dortmund im Sommersemester 1989 gehalten habe. Ich möchte allen danken, die an der Erstellung dieses ersten Skripts beteiligt waren. Mein Dank gilt insbesondere Astrid Baumgart, die sich durch TEX vom Schreiben nicht abschrecken ließ, Ralf Hinze und besonders Jörg Süggel für viele Korrekturhinweise.

Aachen, Juli 1997 Michael Hanus

P.S.: Wer in diesem Skript keine Fehler findet, hat sehr unaufmerksam gelesen. Ich bin für alle Hinweise auf Fehler dankbar, die mir persönlich, schriftlich oder per e-mail (`hanus@informatik.rwth-aachen.de`) mitgeteilt werden.

Inhaltsverzeichnis

Kapitel 0

Einführung

Zum Titel der Vorlesung:

Was sind logische Programmiersprachen?

Eine mögliche Antwort ist in [GM86] angegeben:

> Eine **logische Programmiersprache** ist eine Sprache, deren Programme Axiome in einem wohlverstandenen logischen System (d.h. mit einfacher Semantik, einfachem Deduktionsbegriff und einem Vollständigkeitstheorem) sind und deren operationale Semantik *Deduktion* in diesem System ist.

Dies umfaßt

- relationale Sprachen (basierend auf Hornklausellogik) wie Prolog und
- funktionale Sprachen (basierend auf Gleichungslogik, λ-Kalkül).

In dieser Vorlesung werden wir allerdings logische Programmiersprachen nur als Sprachen auf der Basis der Prädikatenlogik 1. Stufe verstehen.

0.1 Geschichte der Implementierung logischer Programmiersprachen

(siehe auch [Coh88] [Kow88])

1965 Robinson: Resolutionsprinzip zum automatischen Beweisen von Aussagen der Prädikatenlogik 1. Stufe („modus ponens“ kombiniert mit Unifikation) [Rob65]

1969 Green: Lisp-Implementierung der Resolution [Gre69]

1971 Kowalski/Kuehner: Verbesserung der Resolutionsmethode durch „SL-Resolution“ [KK71]

1972 Boyer/Moore: Theorembeweiser mit „structure sharing“ [BM72]

1972-1975 Colmerauer/Roussel: Einschränkung auf Hornklauseln, feste Backtracking-Strategie ⇒ „Prolog“ [Rou75]

1974 Kowalski: “Predicate logic as programming language” [Kow74]. Beginn einer eigenen Forschungsdisziplin mit dem Paradigma $\boxed{\text{Rechnen} = \text{Beweisen}}$

Im folgenden verstärkte Arbeit an effizienteren Implementierungen.

1977 Warren: Übersetzung von Prolog in von-Neumann-Maschinencode (Dec-10-Prolog)

Prolog-X (Clocksin 1985) und New Engine (Warren 1983):

- Überarbeitete Version der Implementierung von 1977
- WAM („Warren Abstract Machine“) ist die Grundlage fast aller heutigen effizienten Prolog-Implementierungen

0.2 Ziele und Inhalt der Vorlesung

In dieser Vorlesung soll ein systematischer Weg von der Semantik logischer Programmiersprachen zu deren effizienten Implementierung aufgezeigt werden, der sich im Überblick wie folgt darstellt:

Deklarative Semantik
↓ (*Herbrand-Modelle, Transformation Tp* [Llo87])
Operationale Semantik
↓ (*Backtracking*)
Interpreter
↓ (*Partielle Auswertung, gute Ideen*)
Compiler

Zusätzlich werden wir auf folgende Aspekte eingehen:

- Implementierung nicht-logischer Konstrukte („Cut“, Datenbankmanipulation etc.)
- Optimierungen durch statische Analyse

Nicht behandelt werden Implementierungsaspekte auf konkreter Hardware (680x0, Risc, spezielle Hardware für Prolog). Hierzu bietet sich die folgende Literatur an:

Herkömmliche Prozessoren: [Pro87] [GMP86]

Risc: [Mil89] [Tay90]

Spezialhardware: [TW84] [KTW+86] [KYAB88]

Für Lisp-Maschinen: [KSKH85]

0.3 Literatur zur Vorlesung

Es folgt eine Auflistung der wichtigsten Literatur zu dieser Vorlesung. Eine vollständige Literaturliste aller zitierten Arbeiten befindet sich am Ende dieses Skripts.

A.V. Aho, R. Sethi, and J.D. Ullman: *Compilers: Principles, Techniques and Tools.*
Addison-Wesley, Reading, Mass., 1985.
Allgemeines, lesenswertes Lehrbuch zum Übersetzerbau.

L. Sterling and E. Shapiro: *The Art of Prolog.*
MIT Press, 1986.
Lesenswertes Lehrbuch über Prolog-Programmiertechniken, beinhaltet auch Aspekte der Implementierung logischer Sprachen mit Prolog (Meta-Interpreter).

F. Kluzniak and S. Szapakowicz: *Prolog for Programmers.*
Academic Press, London, 1985.
Enthält ein Kapitel über Prolog-Implementierungsaspekte und einen kompletten Prolog-Interpreter in Pascal.

D. Maier and D.S. Warren: *Computing with Logic - Logic Programming with Prolog.*
Benjamin/Cummings, 1988.
Einführung in die logische Programmierung, enthält einige Kapitel über die Implementierung von Prolog.

D.H.D. Warren: *Logic Programming and Compiler Writing.*
Software — Practice and Experience, Vol.10, pp. 97-125, 1980.
Artikel über die Vorteile der logischen Programmierung bei der Implementierung von Übersetzern.

J.A. Campbell: *Implementations of Prolog.*
Ellis Horwood, 1984.
Enthält verschiedene Arbeiten zur Prolog-Implementierung.

D.H.D. Warren: *An Abstract Prolog Instruction Set.*
Technical Note 309, SRI International, Stanford, 1983.
Eine knappe Beschreibung der neuen Prolog-Maschine von David Warren.

H. Aït-Kaci: *Warren's Abstract Machine.*
MIT Press, 1991.
Die im Augenblick beste Einführung in die WAM.

Kapitel 1

Theoretische Grundlagen

In diesem Kapitel werden wir die logische Programmiersprache, die dieser Vorlesung zugrunde liegt, definieren (Syntax und Semantik) und ein effizientes Beweisverfahren für diese Sprache angeben. Das Ziel der Vorlesung ist die Entwicklung von effizienten Implementierungen dieses Beweisverfahrens auf herkömmlichen (von Neumann-) Rechnerarchitekturen.

Im folgenden werden wir direkt die Sprache der Hornklausellogik definieren (im Gegensatz zum Buch von Lloyd, in dem zunächst die allgemeine Prädikatenlogik 1. Stufe definiert und diese dann auf Hornklauseln eingeschränkt wird).

Hier werden nur die wichtigsten Ergebnisse dargestellt. Weitere Details findet man in den Büchern von Lloyd [Llo87] und Padawitz [Pad88].

1.1 Syntax logischer Programme

Def.: Ein **logisches Programm** oder **Hornklauselprogramm** besteht aus

- Variablen ($X, Y, Z \in Var$)
- Funktoren ($f/m,\ g/n \in Func$)
- Prädikaten ($p/m,\ q/n \in Pred$)
- den Symbolen ‘,’ (*Konjunktion*) und ‘$\leftarrow$’ (*Implikation*)

und ist wie folgt aufgebaut:

1. Ein **Term** ist entweder
   - eine **Variable** X, wobei $X \in Var$,
   - eine **Konstante** c, wobei $c/0 \in Func$, oder
   - ein **zusammengesetzter Term** $f(t_1, \ldots, t_n)$, wobei $f/n \in Func\,(n > 0)$ und $t_1, \ldots, t_n$ Terme sind.
2. Ein **Literal** hat die Form $p(t_1, \ldots, t_n)$, wobei $p/n \in Pred(n \geq 0)$ und $t_1, \ldots, t_n$ Terme sind.
3. Ein **Ziel** ist eine endliche Folge von Literalen.

4. Eine (**Programm**-) **Klausel** hat die Form $L_0 \leftarrow L_1, \ldots, L_k$ wobei $L_0, L_1, \ldots, L_k$ Literale sind. Im Falle von $k = 0$ wird die Klausel auch **Faktum** genannt.
5. Ein **logisches Programm** ist eine endliche Menge von Klauseln.

Anmerkungen:

- Wir gehen im folgenden immer davon aus, daß es mindestens eine Konstante $c/0 \in Func$ gibt.
- Da in Prolog die Reihenfolge der Klauseln relevant ist, ist ein Prolog-Programm eine **Folge** von Klauseln und keine Menge. Dies ist für die (deklarative) Semantik jedoch nicht relevant.
- In Prolog wird bei Termen unterschieden zwischen allgemeinen Termen, Zahlen und Listenstrukturen. Diese Unterscheidung ist für die Semantik nicht notwendig, bringt aber bei der Implementierung Vorteile. Wir werden später darauf noch einmal eingehen.
- In der logischen Programmierung geht es darum, ein Ziel bzgl. eines gegebenen Programms zu verifizieren. Das zu beweisende Ziel nennt man auch **Anfrage**.

Beispiel: $Var = \{\texttt{E}, \texttt{L}, \texttt{R}, \texttt{RL}\}$, $Func = \{\texttt{[]}/0, \bullet/2\}$, $Pred = \{\texttt{append}/3\}$

Klauseln:

```
append([],L,L) ←
append([E|R],L,[E|RL]) ←   append(R,L,RL)
```

(hierbei steht `[T1|T2]` für den Term `•(T1,T2)`)

1.2 Semantik logischer Programme

Die Semantik basiert auf den Begriffen Interpretation, Variablenbelegung, Gültigkeit und Modell. Eine Interpretation ist die Zuordnung der Symbole in einem Programm zu entsprechenden mathematischen Strukturen. Eine Variablenbelegung ordnet jeder Variablen einen Wert zu, und eine Klausel heißt gültig, wenn sie eine richtige Implikation für alle Variablenbelegungen ist. Wir interessieren uns nur für Interpretationen, in denen alle Klauseln eines Programms gültig sind. Diese heißen auch Modelle. Es folgt eine formale Definition dieser Begriffe:

Def.: Eine **Interpretation** für ein logisches Programm ist eine Menge U zusammen mit einer Zuordnung δ, wobei:

- U heißt **Universum** oder **Trägermenge** der Interpretation.
- Jeder Konstanten $c/0 \in Func$ wird ein Wert im Universum $\delta_c \in U$ zugeordnet.
- Jedem Funktor $f/n \in Func$ wird eine n-stellige Funktion $\delta_{f/n} : \underbrace{U \times \ldots \times U}_{n-mal} \rightarrow U$ zugeordnet.

- Jedem Prädikat $p/n \in Pred$ wird eine n-stellige Relation $\delta_{p/n} \subseteq \underbrace{U \times \ldots \times U}_{n-mal}$ zugeordnet.

Def.: Eine **Variablenbelegung** $v : Var \to U$ ordnet jeder Variablen ein Element im Universum U zu.

Def.: Sei $I = (U, \delta)$ eine Interpretation und v eine Variablenbelegung. Der **Wert eines Terms** t wird mit $I_v(t)$ bezeichnet und ist wie folgt definiert:

- Für alle Variablen $X \in Var$ ist $I_v(X) := v(X)$
- Für alle Konstanten c ist $I_v(c) := \delta_c \ (\in U)$
- Für alle zusammengesetzten Terme $f(t_1, \ldots, t_n)$ ist

$$I_v(f(t_1, \ldots, t_n)) := \delta_{f/n}(I_v(t_1), \ldots, I_v(t_n))$$

Gültigkeit von Literalen, Zielen und Klauseln:

1. *Gültigkeit bzgl. einer Interpretation und einer festen Variablenbelegung v:*
   $I, v \models p(t_1, \ldots, t_n)$ (Literal ist gültig), wenn $(I_v(t_1), \ldots, I_v(t_n)) \in \delta_{p/n}$
   $I, v \models L_1, \ldots, L_k$ (Ziel ist gültig), wenn $I, v \models L_i$ für $i = 1, \ldots, k$
   $I, v \models L_0 \leftarrow L_1, \ldots, L_k$ (Klausel ist gültig), wenn gilt: Wenn $I, v \models L_1, \ldots, L_k$, dann ist auch $I, v \models L_0$
2. *Gültigkeit bzgl. einer Interpretation:*
   $I \models L$ (L ist Literal), wenn $I, v \models L$ für alle Variablenbelegungen v
   $I \models L_1, \ldots, L_k$ wenn $I, v \models L_1, \ldots, L_k$ für alle Variablenbelegungen v
   $I \models L_0 \leftarrow L_1, \ldots, L_k$ wenn $I, v \models L_0 \leftarrow L_1, \ldots, L_k$ für alle Variablenbelegungen v
3. *Modell:* Sei C ein logisches Programm. Eine Interpretation I heißt **Modell** für C, wenn für alle Klauseln $L \leftarrow G \in C$ gilt: $I \models L \leftarrow G$
4. *Gültigkeit:* Ein Ziel G heißt **gültig bezüglich eines logischen Programms** C, in Zeichen $C \models G$, wenn für jedes Modell I für C gilt: $I \models G$. Man sagt auch: G ist eine **logische Konsequenz** von C.

Der unter 4. definierte Begriff ist die formale Definition der **deklarativen Semantik** von logischen Programmen. Logische Konsequenzen sind Aussagen, die immer aus den im Programm formulierten Aussagen folgen (unabhängig vom gewählten Modell). Die Aufgabe einer Implementierung für eine logische Programmiersprache ist die Automatisierung der Überprüfung, ob ein Ziel eine logische Konsequenz aus dem gegebenen Programm ist.

Beispiel: Programm

```
p(a) ←
p(b) ←
q(X) ← p(X)
```

Dann sind jeweils
`p(a)`
`p(b)`
`q(a)`
`q(b)`
logische Konsequenzen aus dem Programm.

Kommen in einem Ziel noch **Variablen** vor, dann soll die Implementierung nicht überprüfen, ob genau dieses Ziel eine logische Konsequenz ist, sondern sie soll Werte finden, die man für die Variablen einsetzen kann, so daß das so modifizierte Ziel eine logische Konsequenz ist. Die Werte dieser Variablen heißen **(korrekte) Antwort**. (Variablen in Anfragen sind also existenzquantifiziert, während Variablen in Klauseln allquantifiziert sind.)

Beispiel: Gegeben sei obiges Programm und die Anfrage `q(X)`. Dann sind sowohl `X` = `a` wie auch `X` = `b` korrekte Antworten.

Die **zentrale Frage** für die Implementierung einer logischen Programmiersprache ist: Wie kann man (effizient) automatisch nachweisen, ob ein Ziel gültig bzgl. eines logischen Programms ist?

Zu diesem Zweck wurden spezielle Beweisverfahren entwickelt, nach denen heutige Implementierungen arbeiten.

1.3 Beweisverfahren für logische Programme

Um die Gültigkeit eines Ziels bzgl. eines logischen Programms nachzuweisen, könnte man systematisch alle logischen Konsequenzen aus dem Programm aufzählen und dabei prüfen, ob das gegebene Ziel darunter ist. Dies ist natürlich extrem ineffizient. Daher arbeiten effiziente Beweisverfahren **zielgerichtet**: Sie versuchen Beweise „rückwärts“ zu konstruieren („backward chaining“), indem das gegebene Ziel unter Berücksichtigung des Programms in ein anderes, möglichst einfacheres Ziel transformiert wird. Dabei wird in jedem Transformationsschritt eine Klausel aus dem Programm verwendet, deren linke Seite zu einem Literal in dem Ziel „paßt“, und dann wird dieses Literal durch die rechte Seite der passenden Klausel ersetzt. Um die Anzahl der passenden Klauseln klein zu halten, werden Unifikatoren berechnet, die wir zunächst definieren.

Def.: Eine **Substitution** σ ist eine Abbildung $\sigma: Var \rightarrow Term$, die jeder Variablen einen Term zuordnet. Eine Substitution σ kann in natürlicher Weise auf Termen fortgesetzt werden durch:

- Für alle Konstanten c sei $\sigma(c) := c$
- Für alle zusammengesetzten Terme $f(t_1, \ldots, t_n)$ sei

$$\sigma(f(t_1, \ldots, t_n)) := f(\sigma(t_1), \ldots, \sigma(t_n))$$

Analog: Fortsetzung auf Literalen, Zielen und Klauseln. Ein Term (Literal, Ziel, Klausel) t heißt **Instanz** von t', wenn eine Substitution σ existiert mit $\sigma(t') = t$.

Substitutionen verändern also nur die Variablen in einem Term. Da wir es in der Regel nur mit endlichen Substitutionen zu tun haben, d.h. Substitutionen, die nur endlich viele Variablen verändern, notieren wir Substitutionen in der Form $\sigma = \{X \mapsto \sigma(X) \mid X \in Var \text{ und } X \neq \sigma(X)\}$.

Beispiel: Sei $\sigma = \{\texttt{X} \mapsto \texttt{a},\ \texttt{Y} \mapsto \texttt{f(b)}\}$, dann ist

$$\sigma(\texttt{g(X,Y)}) = \texttt{g(a, f(b))}$$

$$\sigma(\texttt{h(a,X,Z)}) = \texttt{h(a,a,Z)}$$

Def.: Die Terme (Literale, Ziele, Klauseln) t_1 und t_2 heißen **Varianten**, wenn Substitutionen σ_1 und σ_2 existieren mit $\sigma_1(t_1) = t_2$ und $\sigma_2(t_2) = t_1$.

Varianten unterscheiden sich nur durch die Namen der Variablen.

Def.: Seien t_1 und t_2 Terme (Literale). Eine Substitution σ heißt **Unifikator** für t_1 und t_2, wenn $\sigma(t_1) = \sigma(t_2)$. In diesem Fall heißen t_1 und t_2 **unifizierbar**.

Ein Unifikator σ heißt **allgemeinster Unifikator** („most general unifier“, „mgu“) für t_1 und t_2, wenn für jeden Unifikator σ' für t_1 und t_2 gilt: Es existiert eine Substitution γ mit $\sigma' = \gamma \circ \sigma$ (d.h. $\sigma'(t) = \gamma(\sigma(t))$ für alle Terme t).

Allgemeinste Unifikatoren sind bis auf Variablenumbenennung eindeutig bestimmt. Robinson hat einen Algorithmus zur Berechnung eines allgemeinsten Unifikators gefunden (siehe [Rob65]). Diesen und andere Unifikationsalgorithmen werden wir in Kapitel 2.3 kennenlernen.

Nun können wir ein effizienteres Beweisverfahren definieren:

Def.: Sei $G = L_1, \ldots, L_m, \ldots, L_k$ ein Ziel und $C = A_0 \leftarrow A_1, \ldots, A_q$ eine Klausel. Dann heißt G' **abgeleitet** aus G und C mit allgemeinstem Unifikator σ, wenn gilt:

- L_m ist das aus G **ausgewählte Literal**
- σ ist ein allgemeinster Unifikator für L_m und A_0
- $G' = \sigma(L_1, \ldots, L_{m-1}, A_1, \ldots, A_q, L_{m+1}, \ldots, L_k)$

Anmerkung: Wir gehen davon aus, daß eine Regel existiert, die genau ein Literal aus einem Ziel auswählt. Diese Regel heißt **Auswahlregel** („selection function“). In Prolog lautet die Auswahlregel: „Wähle das am weitesten links stehende Literal“ ($m = 1$).

Def.: Eine **SLD-Ableitung** für ein Ziel G ist eine Folge von Zielen $G_1, G_2, \ldots$ mit:

- $G = G_1$
- G_{i+1} ist abgeleitet aus G_i und einer *Variante* C_i einer Programmklausel mit allgemeinstem Unifikator σ_i, wobei C_i keine Variablen enthält, die in der Ableitung bis G_i vorgekommen sind.

Eine SLD-Ableitung heißt **erfolgreich**, wenn sie endlich ist und das letzte Ziel G_n leer ist. In diesem Fall heißt die Einschränkung der Substitution $\sigma_{n-1} \circ \ldots \circ \sigma_1$ auf die Variablen in G die **berechnete Antwort**.

SLD steht für „**L**inear resolution with **S**election function for **D**efinite clauses“. Die Verwendung von Varianten ist wesentlich für die Vollständigkeit der SLD-Resolution.

Beispiel: Programm

`p(a)` $\leftarrow$
`p(b)` $\leftarrow$
`q(Y)` $\leftarrow$ `p(Y)`

Auswahlregel: $m = 1$

SLD-Resolution für das Ziel `q(X), p(b)`:

`q(X), p(b)`
  $\vdash$ (3. Klausel) $\sigma_1 = \{\texttt{Y} \mapsto \texttt{X}\}$
`p(X), p(b)`
  $\vdash$ (1. Klausel) $\sigma_2 = \{\texttt{X} \mapsto \texttt{a}\}$
`p(b)`
  $\vdash$ (2. Klausel) $\sigma_3 = \{\}$
$\emptyset$

Berechnete Antwort: $\sigma = \{\texttt{X} \mapsto \texttt{a}\}$

Mittels der SLD-Resolution kann die Gültigkeit von Zielen bewiesen werden:

Satz: Sei P ein Programm und G ein Ziel. Jede berechnete Antwort für G bzgl. P ist eine korrekte Antwort.

Die Umkehrung gilt nur eingeschränkt, da wegen der mgu's in jedem Schritt nur allgemeinste Antworten berechnet werden:

Satz: Sei P ein Programm und G ein Ziel. Zu jeder korrekten Antwort σ für G bzgl. P existiert eine berechnete Antwort γ und eine Substitution ϕ mit $\sigma = \phi \circ \gamma$.

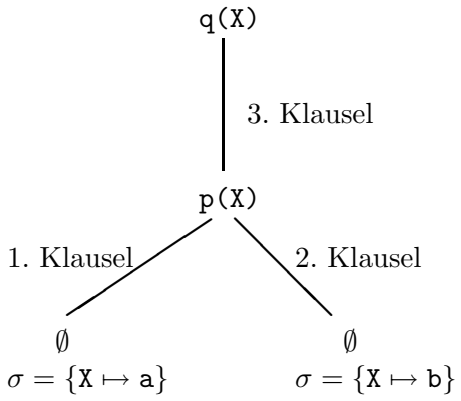
Da die beiden Sätze für jede beliebige Auswahlregel gelten, können wir uns im folgenden auf eine festlegen. Wir wählen die in Prolog übliche ($m = 1$). Damit haben wir eine korrekte und vollständige Beweismöglichkeit für logische Konsequenzen. Vollständig ist dies aber nur, wenn alle SLD-Ableitungen gleichzeitig untersucht werden. Eine präzise Formulierung dieses Sachverhalts ist mittels SLD-Bäumen möglich:

Def.: Sie P ein Programm und G ein Ziel. Ein **SLD-Baum** für G bzgl. P ist ein Baum, dessen Knoten mit (evtl. leeren) Zielen markiert sind, wobei:

- Die Wurzel ist mit G markiert.

- Ist ein Knoten mit dem Ziel $L_1, \ldots, L_k$ $(k > 0)$ markiert, dann existiert für jede Programmklausel, mit der dieses Ziel zu einem Ziel G' abgeleitet werden kann, ein Sohn dieses Knotens, der mit G' markiert ist.
- Knoten, die mit leeren Zielen markiert sind, haben keine Söhne.

Beispiel: Programm wie oben, Ziel: `q(X)`
SLD-Baum:



Eine **Beweisstrategie** ist eine Strategie zum Durchsuchen eines SLD-Baumes. Jeder Zweig (Weg von der Wurzel zu einem Blatt) im SLD-Baum entspricht einer SLD-Ableitung. Daher: Ein **Breitendurchlauf** durch den SLD-Baum ist eine vollständige Beweisstrategie für logische Konsequenzen.

Nachteil: Hoher Verwaltungsaufwand, da der Baum schnell in die Breite wächst (ein Binärbaum hat in der 10. Stufe schon 1024 Knoten!).

Aus diesem Grund verzichtet man in Prolog auf die Vollständigkeit der Beweisstrategie zugunsten der Effizienz und macht einen **Tiefendurchlauf** durch den SLD-Baum. Hierzu müssen folgende Vereinbarungen getroffen werden:

1. Alle Klauseln im Programm sind geordnet in ihrer textuellen Reihenfolge.
2. Die Söhne jedes Knotens im SLD-Baum sind geordnet.
3. Wird ein Ziel G zu G_1 mit der Klausel C_1 bzw. zu G_2 mit der Klausel C_2 abgeleitet, dann steht G_1 im SLD-Baum *links* von G_2, wenn C_1 *textuell vor* C_2 steht.

Dann führt Prolog einen Links-Rechts-Tiefendurchlauf durch den SLD-Baum aus. Dies ist die **operationale Semantik** von Prolog.

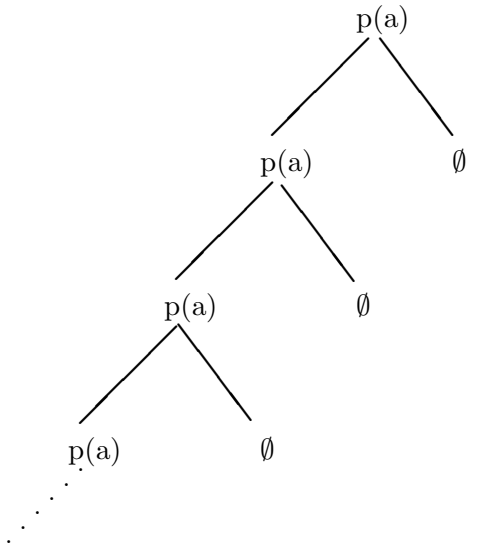
Probleme ergeben sich bei dieser operationalen Semantik, wenn unendliche Zweige links von Erfolgszweigen im SLD-Baum vorhanden sind. In diesem Fall terminiert das Prolog-System nicht und kann die korrekten Lösungen nicht finden.

Beispiel: Programm:

`p(a)` ← `p(a)`
`p(a)` ←

Ziel: `p(a)`

SLD-Baum:



Der Tiefendurchlauf endet nicht, d.h. das Ziel kann nicht bewiesen werden, obwohl es eine logische Konsequenz ist.

Vorteile der operationalen Semantik von Prolog:

- Es handelt sich um ein deterministisches Beweisverfahren, d.h. es ist eindeutig definiert, wie ein Ziel bzgl. eines Programms bewiesen wird. Daher sind Prädikate mit Seiteneffekten (Ein-/Ausgabe) möglich, und deren Wirkung ist eindeutig definiert.
- Das Verfahren kann effizient mit Hilfe von Stacks implementiert werden. Dabei wird bei jedem Schritt vom Vater- zu einem Sohnknoten ein Element (der neue Sohnknoten) auf den Stack gelegt („push node“), und bei jedem Schritt von einem Sohnknoten zum Vaterknoten wird ein Element vom Stack weggenommen („pop node“). Der aktuelle Stackzustand repräsentiert somit den Weg von der Wurzel zum aktuellen Knoten im SLD-Baum. Weitere Details findet man im Rest dieser Vorlesung.

Kapitel 2

Interpreter für PROLOG

Allgemeine Literatur zu diesem Kapitel: [vE84] [Fuc84] [Coh85]

Ein **Interpreter** nimmt ein Programm (und die Eingaben an dieses Programm) als Eingabe und arbeitet es gemäß der operationalen Semantik ab. Ein Interpreter für Prolog nimmt also das logische Programm und eine Anfrage als Eingabe und versucht die Anfrage gemäß der SLD-Resolution zu beweisen. In diesem Kapitel werden wir zunächst Prolog-Interpreter durch mathematische Maschinen spezifizieren, da dies eine einfache und präzise Beschreibungsmethode ist. Anschließend werden wir auf spezielle Details bei der Implementierung von Prolog-Interpretern eingehen (Unifikationsalgorithmen, Speicherverwaltung).

2.1 Operationale Semantik von Prolog

In der Literatur gibt es verschiedene Ansätze zur Beschreibung der operationalen Semantik von Prolog. Prinzipiell ist dies ja ein Tiefendurchlauf durch den SLD-Baum, aber viele Ansätze versuchen, die Semantik direkt und nicht über den Umweg des SLD-Baums zu beschreiben. In [Nil84] und [AB87] wird VDM („Vienna Definition Method“) zur Spezifikation benutzt. Die Beschreibung in [DF87] basiert auf SLD-Bäumen, während [DM88] und [JM84] mathematische Maschinen benutzen (die Beschreibung in [Bör90a, Bör90b] ist dazu sehr ähnlich; sie basiert allerdings auf dem Formalismus der dynamischen Algebren). Wenn wir diese Maschinen in einer gewöhnlichen Programmiersprache (z.B. Pascal) implementieren, erhalten wir einen Interpreter. Ein weiterer Vorteil ist, daß sich die Semantik nicht-logischer Konstrukte (Cut, Datenbank-Operationen) hiermit sehr leicht spezifizieren läßt. Daher werden wir diese Maschinenbeschreibungen im folgenden verwenden.

Um die Wertebereiche eines Interpreters zu beschreiben, definieren wir die folgenden **syntaktischen Kategorien** von Prolog (dies entspricht einer **abstrakten Syntax** für Prolog):

- Var ::= {Menge von Variablennamen}
- $Func$ ::= {Menge von Funktornamen}
- $Pred$ ::= {Menge von Prädikatnamen}
- $Term ::= Var \cup Func(Term^*)$

- $Lit ::= Pred(Term^*)$
- $Goal ::= Lit^*$
- $Clause ::= Lit \leftarrow Goal$
- $Program ::= Clause^*$

Anmerkung: Manche Autoren definieren ein Programm als Folge von Klauseln **zusammen** mit einer Anfrage. Die meisten Prolog-Systeme übersetzen jedoch nur Klauselfolgen.

Notationen für syntaktische Elemente:

- Folgen von Termen werden mit Kommata aufgeschrieben: $f(a, b, c) \in Func(Term^*)$.
- „true“ steht für die leere Folge von Literalen.
- $l \wedge g$ steht für das Ziel, dessen erstes Literal l ist und dessen restliches Ziel g ist.
- $\wedge$ steht auch für die Konkatenation von Zielen.
- Wenn nichts anderes gesagt, werden Folgen (von Klauseln o.ä.) mit dem Symbol :: aufgeschrieben und die leere Folge wird mit nil bezeichnet.
- Für eine Folge s gibt $length(s)$ die Anzahl der Elemente in s an.

Die Interpreterbeschreibungen beinhalten außerdem noch die folgenden wichtigen Funktionen:

$Subst = Var \rightarrow Term$ (Menge aller *Substitutionen*, d.h. Abbildungen von Variablen in Terme)

$Rename = Var \rightarrow Var$ (Menge aller *Variablenumbenennungen*, d.h. injektiven Abbildungen von Variablen in Variablen)

Zur einfachen Umbenennung von Variablen:

- Unterteile Var in abzählbar viele paarweise disjunkte Mengen: $Var = Var_0 \cup Var_1 \cup Var_2 \cup \ldots$
- Alle Variablen in Programmklauseln und der Anfrage an ein Programm sind aus Var_0.
- Für alle $n \in Nat$ existiert ein $\rho_n \in Rename$ mit $\rho_n : Var_0 \rightarrow Var_n$ (ρ_0 ist die Identität).
- Vorstellung: Alle Variablen sind indiziert mit natürlichen Zahlen und die ρ_i verändern diesen Index. Den Index 0 lassen wir in den folgenden Beispielen meistens weg.

$MGU : Lit \times Lit \rightarrow Subst \cup \{fail\}$

liefert einen allgemeinsten Unifikator, falls die Argumente unifizierbar sind, und sonst „$fail$“. Verschiedene Algorithmen hierfür werden später (Kapitel 2.3) vorgestellt.

Nichtdeterministischer Interpreter ohne Ausgaben:

Zustandsraum des Interpreters:	$State = Goal \times Nat \times Program$
Eingabe:	Programm P, Anfrage G
Startzustand:	$(G, 0, P)$
Endzustand:	$(true, n, P)$ (n beliebig)

Zustandsübergang ($\rightarrow$ ist eine Relation auf $State \times State$):

$$(l \wedge g, n, P) \rightarrow (\sigma(\rho_{n+1}(b) \wedge g), n+1, P),$$

falls in P eine Klausel $a \leftarrow b$ vorkommt mit $MGU(l, \rho_{n+1}(a)) = \sigma \neq fail$

Anmerkungen:

- Die erste Komponente des Zustands enthält das noch zu beweisende Ziel. Die zweite Komponente enthält die Nummer des Ableitungsschritts und hat den Zweck, die Variablen der angewendeten Klausel in jedem Schritt so umzubenennen, daß es keine Namenskonflikte mit vorher vorkommenden Variablen gibt.
- Der Zustandsübergang ist wie ein Ableitungsschritt der SLD-Resolution mit Auswahlregel $m = 1$ definiert. Daher beschreibt dieser Interpreter genau die Implementierung der SLD-Resolution.
- Jede Zustandsübergangsfolge $(G, 0, P) \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow (true, n, P)$ entspricht einer SLD-Ableitung von G bzgl. P.

Beispiel: Programm P:

$\texttt{p(a)} \leftarrow$
$\texttt{p(b)} \leftarrow$
$\texttt{q(X)} \leftarrow \texttt{p(X)}$

Anfrage:

`q(X)`

Zustandsfolge:

(`q(X)`, 0, P)
$\rightarrow$ (`p(X)`, 1, P) (da $MGU(\texttt{q(X)}, \texttt{q(X1)}) = \{\texttt{X1} \mapsto \texttt{X}\}$)
$\rightarrow$ (`true`, 2, P) (da $MGU(\texttt{p(X)}, \texttt{p(b)}) = \{\texttt{X} \mapsto \texttt{b}\}$)

Ausgegeben werden müßte nun $\{\texttt{X} \mapsto \texttt{b}\}$, was aber im Interpreter noch nicht vorgesehen ist. Dazu kommen wir noch später.

Prolog liegt jedoch ein deterministisches Verfahren zugrunde. Daher geben wir nun einen deterministischen Interpreter an, der einen Links-Rechts-Tiefendurchlauf durch den SLD-Baum mit Hilfe eines Backtracking-Verfahrens simuliert:

Interpreter mit Backtracking:

Zustandsraum:	$State = (Goal \times Program)^* \times Program$
Eingabe:	Programm P, Anfrage G
Startzustand:	$((G, P) :: nil, P)$
Endzustand:	(nil, P)

Zustandsübergangsregeln:

1. Anwendung der nächsten Klausel (falls möglich):

$$\begin{aligned}((l \wedge g, (a \leftarrow b) :: cs) :: st, P) \quad &\rightarrow \quad ((\sigma(\rho_{n+1}(b) \wedge g), P) :: (l \wedge g, cs) :: st, P) \\ &\quad\ \text{falls } MGU(l, \rho_{n+1}(a)) = \sigma \neq fail \\ &\rightarrow \quad ((l \wedge g, cs) :: st, P) \\ &\quad\ sonst\end{aligned}$$

   wobei $n = length(st)$.

2. Keine weitere Klausel ist anwendbar: Backtracking:

$$((g, nil) :: st, P) \quad \rightarrow \quad (st, P)$$

3. Die Anfrage ist bewiesen: Backtracking:

$$((true, cs) :: st, P) \quad \rightarrow \quad (st, P)$$

Anmerkungen:

- Die erste Komponente des Zustands ist ein Stack, der alle noch zu beweisenden Ziele enthält.
- Im Zustand sind zu jedem Ziel die Klauseln gespeichert, die noch nicht zum Beweis des ersten Literals des Ziels ausprobiert worden sind.
- Bei erfolgreicher Unifikation vergrößert sich der Stack um 1, da der Beweis desselben Ziels mit einer anderen Klausel zurückgestellt und auf den Stack gelegt wird. Daher gibt die Länge des Stacks die Ableitungstiefe+1 an und somit kann die Nummer des Ableitungsschritts berechnet werden und muß nicht explizit im Zustand gespeichert werden (wie beim vorherigen Interpreter).
- Beim Zustandsübergang nach der 3. Regel ist ein erfolgreicher Beweis gefunden und die Antwort kann ausgegeben werden (s.u.).

Beispiel: Programm P:

```
p(a) ←
p(b) ←
q(X) ← p(X)
r(b) ←
```

Anfrage:

```
q(X) ∧ r(X)
```

Zustandsfolge (P_i bezeichnet das Programm P ohne die ersten i Klauseln):

$((\texttt{q(X)}\wedge\texttt{r(X)}, P) :: nil, P)$
$\rightarrow \quad ((\texttt{q(X)}\wedge\texttt{r(X)}, P_1) :: nil, P)$
$\rightarrow \quad ((\texttt{q(X)}\wedge\texttt{r(X)}, P_2) :: nil, P)$
$\rightarrow \quad ((\texttt{p(X)}\wedge\texttt{r(X)}, P) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$
$\rightarrow \quad ((\texttt{r(a)}, P) :: (\texttt{p(X)}\wedge\texttt{r(X)}, P_1) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$
$\rightarrow^4 \quad ((\texttt{r(a)}, nil) :: (\texttt{p(X)}\wedge\texttt{r(X)}, P_1) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$
$\rightarrow \quad ((\texttt{p(X)}\wedge\texttt{r(X)}, P_1) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$ (Backtracking wegen Fehlschlag)
$\rightarrow \quad ((\texttt{r(b)}, P) :: (\texttt{p(X)}\wedge\texttt{r(X)}, P_2) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$
$\rightarrow^3 \quad ((\texttt{r(b)}, P_3) :: (\texttt{p(X)}\wedge\texttt{r(X)}, P_2) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$
$\rightarrow \quad ((\texttt{true}, P) :: (\texttt{r(b)}, nil) :: (\texttt{p(X)}\wedge\texttt{r(X)}, P_2) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$
$\rightarrow \quad ((\texttt{r(b)}, nil) :: (\texttt{p(X)}\wedge\texttt{r(X)}, P_2) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$ (Backtracking nach erfolgreichem Beweis)
$\rightarrow \quad ((\texttt{p(X)}\wedge\texttt{r(x)}, P_2) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$ (Backtracking wegen Fehlschlag)
$\rightarrow^2 \quad ((\texttt{p(X)}\wedge\texttt{r(X)}, nil) :: (\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$
$\rightarrow \quad ((\texttt{q(X)}\wedge\texttt{r(X)}, P_3) :: nil, P)$ (Backtracking wegen Fehlschlag)
$\rightarrow \quad ((\texttt{q(X)}\wedge\texttt{r(X)}, nil) :: nil, P)$
$\rightarrow \quad (nil, P)$ (Backtracking wegen Fehlschlag)

Implementierung des Interpreters:

Wenn wir diesen Interpreter in einer vorhandenen Sprache (z.B. Pascal) implementieren, haben wir ein Prolog-System realisiert. Dabei kann die zweite Komponente des Zustands als globale Variable realisiert werden (diese Komponente wird ja bei der Abarbeitung nicht verändert). Die zweite Komponente jedes Stackelements kann als Zeiger in das eingegebene Programm realisiert werden. Aufwendig ist nur noch die Speicherung aller während der Laufzeit entstehenden Ziele auf dem Stack. Dies kann, wie wir jetzt sehen werden, auch noch wesentlich verbessert werden.

2.2 Interpreter mit Structure Sharing

Wenn wir den letzten Interpreter genauer analysieren, dann kann man folgendes feststellen: Auf dem Stack stehen nur Ziele, die aus Klauselrümpfen oder der Anfrage durch Substitution von Variablen entstanden sind. Solche Ziele lassen sich auch als Paar (Ziel, Substitution) repräsentieren, wobei das Ziel als Zeiger in eine Programmklausel oder Anfrage realisiert werden kann. Eine solche Repräsentation nennt man **„structure sharing“** [BM72]. Diese Repräsentation wird auch für die Umbenennung von Klauseln verwendet, so daß ein Klauselrumpf nun in der Form (Ziel, Nummer) dargestellt wird. Damit erhalten wir den folgenden Interpreter:

Interpreter mit „structure sharing“:

Zustandsraum:	$State = (RenamedGoal \times Subst \times Program)^* \times Program$
	wobei $RenamedGoal = (Goal \times Nat)^*$
Eingabe:	Programm P, Anfrage G
Startzustand:	$(((G, 0) :: nil, id, P) :: nil, P)$ (id ist die identische Substitution)
Endzustand:	(nil, P)

Zustandsübergangsregeln:

1. Anwendung der nächsten Klausel (falls möglich):
   $(((l \wedge g, m) :: rg, \psi, (a \leftarrow b) :: cs) :: st, P)$

   $\rightarrow (((b, n+1) :: (g, m) :: rg, \sigma \circ \psi, P) :: st_1, P)$
   falls $MGU(\psi(\rho_m(l)), \rho_{n+1}(a)) = \sigma \neq fail$
   $\rightarrow (st_1, P)$, sonst

   wobei $n = length(st)$ und $st_1 = ((l \wedge g, m) :: rg, \psi, cs) :: st$

2. Keine weitere Klausel anwendbar: Backtracking:

$$(((l \wedge g, m) :: rg, \psi, nil) :: st, P) \rightarrow (st, P)$$

3. Teilziel ist bewiesen: Mache weiter mit nächstem Teilziel:

$$(((true, m) :: rg, \psi, cs) :: st, P) \rightarrow ((rg, \psi, P) :: st, P)$$

4. Anfrage ist bewiesen: Backtracking:

$$((nil, \psi, cs) :: st, P) \rightarrow (st, P)$$

Implementierungshinweise:

- Das aktuell zu beweisende Ziel ist hier eine Liste, deren Elemente als ein Zeiger in eine Programmklausel oder Anfrage und einen Index zur Umbenennung von Variablen realisiert werden können. Bei der erfolgreichen Unifikation im 1. Fall muß diese Liste (rg) nicht kopiert werden, sondern kann durch einen Zeiger realisiert werden.
- Ebenso können die 3. Komponenten der Stackelemente durch Zeiger in das Programm und das Programm als globale Variable realisiert werden.
- Bei Anwendung der 4. Regel kann die Antwort ψ ausgegeben werden (evtl. nur die Einschränkung auf die in der Anfrage vorkommenden Variablen) (s.u.).
- Um beim Beweis eines Literals nicht die gesamte Klauselliste zu durchsuchen, kann man die Klauselliste nach Prädikatbezeichnern (Name und Stelligkeit) sortieren. Beim Beweis eines Literals muß man nur noch die Teilliste der Klauseln mit gleichem Prädikatbezeichner ausprobieren.

Berechnung der Ausgabe des Interpreters:

Ausgabe = Folge von Substitutionen (von erfolgreichen Beweisen)

Ein Problem ergibt sich im Falle von unendlichen Berechnungen, da es dann zu unendlichen Folgen von Substitutionen kommen kann. Eine exakte Lösung dieses Problems erhält man durch Benutzung von Domain-Strukturen zur Definition der Semantik von unendlichen Berechnungen (→ „Semantik von Programmiersprachen“). Wir werden diese Lösung im folgenden ganz kurz beschreiben.

Sei $Subst^+$ die Menge der endlichen Listen, die mit *nil* oder $\perp$ (undefiniert, Nichtterminierung) enden. Mache daraus eine *cpo* (*complete partial order*) durch $\perp \sqsubseteq nil, \perp \sqsubseteq X :: Y$ und erweitere diese Relation monoton auf $Subst^+$ (z.B. ist $\sigma_1 :: \perp \sqsubseteq \sigma_1 :: \sigma_2 :: \sigma_3 :: nil$). Dann konstruiere $Subst^\infty$ aus $Subst^+$ durch Hinzunahme aller unendlichen Listen von Substitutionen, d.h. $Subst^\infty$ ist Lösung der *Bereichsgleichung*

$$Subst^\infty = (\{nil\} + (Subst \times Subst^\infty))_\perp$$

Dann definieren wir Funktionen $Output_n$, die die partielle Ausgabe des Interpreters nach n Schritten angeben:

$Output_n : State \to Subst^\infty$
mit

$$\begin{array}{lcl} Output_0((st, P)) & = & \perp \\ Output_{n+1}((nil, P)) & = & \text{nil} \\ Output_{n+1}(((nil, \psi, cs) :: st, P)) & = & \psi :: Output_n((st, P)) \quad \text{(4. Regel wurde angewendet)} \end{array}$$

$Output_{n+1}(S) = Output_n(S')$, falls $S \to S'$ durch Anwendung der Regeln 1, 2 oder 3

Ist S_0 der Startzustand für das Programm P und die Anfrage G, dann ist die Ausgabe des Interpreters definiert als der Grenzwert von $Output_n(S_0)$ für $n \to \infty$.

Der Vorteil dieser Definition ist, daß unterschieden wird zwischen Terminierung und Nichtterminierung:

- Erzeugt der Interpreter die Antworten σ_1 und σ_2 und hält dann an, dann ist die Ausgabe $\sigma_1 :: \sigma_2 :: nil$.
- Erzeugt der Interpreter die Antworten σ_1 und σ_2 und terminiert dann nicht, ohne weitere Lösungen zu produzieren, dann ist die Ausgabe $\sigma_1 :: \sigma_2 :: \perp$.
- Erzeugt der Interpreter nacheinander die unendlich vielen Antworten σ_1, σ_2, σ_3, ..., dann ist die Ausgabe die unendliche Liste $\sigma_1 :: \sigma_2 :: \sigma_3 :: \cdots$.

2.3 Unifikationsalgorithmen

Zentrale Operation im Prolog-Interpreter:

> Unifikation zweier Terme mit Berechnung des allgemeinsten Unifikators (Literale werden in diesem Zusammenhang auch als Terme betrachtet)

Dies ist eine Basisoperation in jedem Prolog-System und wird daher immer als gegeben vorausgesetzt. Deshalb werden in diesem Abschnitt verschiedene Algorithmen hierfür angegeben.

Robinsons „klassischer“ Unifikationsalgorithmus (1965):

Prozedur $unify(t_1, t_2)$
Eingabe: Terme t_1 und t_2
Ausgabe: $(bool, \sigma)$, wobei $bool = true$ genau dann, wenn t_1 und t_2 unifizierbar sind.
Falls $bool = true$, dann ist σ ein *mgu* für t_1 und t_2

BEGIN
  IF $<$ t_1 oder t_2 ist eine Variable $>$
  THEN <Sei x der variable Term und t der andere>
    IF $x = t$
    THEN $bool := true; \sigma := \{\}$
    ELSE IF $occur(x, t)$
      THEN $bool := false$
      ELSE $bool := true;\ \sigma := \{x \mapsto t\}$
      FI
    FI
  ELSE <Sei $t_1 = f(x_1, \ldots, x_n)$ und $t_2 = g(y_1, \ldots, y_m)$ $>$
    IF $f \neq g$ OR $n \neq m$
    THEN $bool := false$
    ELSE $k := 0;\ bool := true;\ \sigma := \{\};$
      WHILE $k < n$ AND $bool$
      DO $k := k + 1;$
        $(bool, \sigma') := unify(\sigma(x_k), \sigma(y_k));$
        IF $bool$ THEN $\sigma := \sigma' \circ \sigma$
        FI
      OD
    FI
  FI
  $return(bool, \sigma)$
END

Anmerkungen:

- Der Original-Algorithmus von Robinson unifiziert beliebig viele Terme gleichzeitig und benutzt sog. „disagreement sets“ in jedem Schritt. Der obige Algorithmus ist schon angepaßt an die Unifikation zweier Terme.
- ***occur*(x, t)** ist genau dann wahr, wenn die Variable x im Term t vorkommt. Ohne diese Bedingung könnten bei der Unifikation zyklische Terme entstehen (die Terme x und $f(x)$ sind nicht unifizierbar). Diese Bedingung heißt **Vorkommenstest** (**occur check**). Ohne Vorkommenstest ist der Unifikationsalgorithmus nicht korrekt. Da dieser Test jedoch einen Durchlauf durch den Term t erfordert, wird in vielen Prolog-Implementierungen auf den Test verzichtet. Da dies aber zu einem fehlerhaften Beweisverfahren führt, sollte dies nicht gemacht werden. Als Alternative könnte man den Vorkommenstest wahlweise abschalten oder durch Voranalyse des Programms herausfinden, an welchen Stellen auf den Test verzichtet werden kann (vgl. Kapitel 6 über statische Analysemethoden).

Beispiel: $unify($`p(1, A, f(g(X)))`$,$`p(X, f(Y), f(Y))`$)

$\rightarrow unify(\mathtt{1}, \mathtt{X}) = (true, \{\mathtt{X} \mapsto \mathtt{1}\})$
$\rightarrow unify(\mathtt{A}, \mathtt{f(Y)}) = (true, \{\mathtt{A} \mapsto \mathtt{f(Y)}\})$

$\rightarrow unify(\mathtt{f(g(1))}, \mathtt{f(Y)})$
$\quad \rightarrow unify(\mathtt{g(1)}, \mathtt{Y}) = (true, \{\mathtt{Y} \mapsto \mathtt{g(1)}\})$
$\quad = (true, \{\mathtt{Y} \mapsto \mathtt{g(1)}\})$

$= (true, \{\mathtt{X} \mapsto \mathtt{1},\ \mathtt{A} \mapsto \mathtt{f(g(1))},\ \mathtt{Y} \mapsto \mathtt{g(1)}\})$

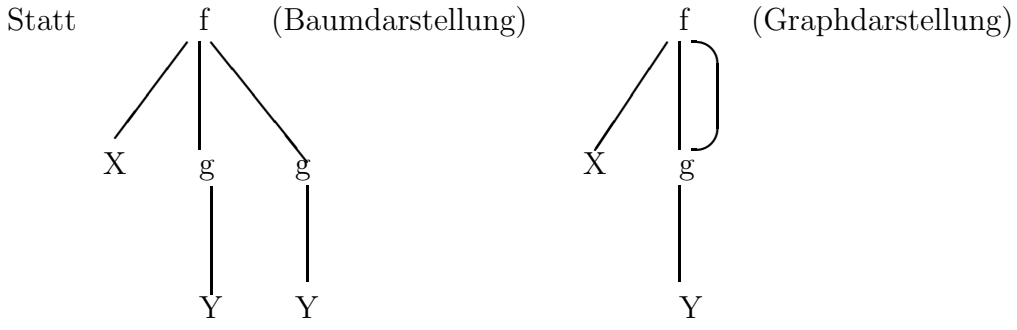
Beispiel: $unify(\mathtt{p(X, X)}, \mathtt{p(Y, f(Y))})$
$\rightarrow unify(\mathtt{X}, \mathtt{Y}) = (true, \{\mathtt{X} \mapsto \mathtt{Y}\})$
$\rightarrow unify(\mathtt{Y}, \mathtt{f(Y)}) = (false, \_)$, weil $occur(\mathtt{Y}, \mathtt{f(Y)}) = true$
$= (false, \_)$

Beispiel: $unify(\mathtt{p(X_1, X_2, X_3)}, \mathtt{p(f(X_0, X_0), f(X_1, X_1), f(X_2, X_2))})$
$\rightarrow unify(\mathtt{X_1}, \mathtt{f(X_0, X_0)}) = (true, \{\mathtt{X_1} \mapsto \mathtt{f(X_0, X_0)}\})$
$\rightarrow unify(\mathtt{X_2}, \mathtt{f(f(X_0, X_0), f(X_0, X_0))})$
$\quad = (true, \{\mathtt{X_2} \mapsto \mathtt{f(f(X_0, X_0), f(X_0, X_0))}\})$
$\rightarrow unify(\mathtt{X_3}, \mathtt{f(f(f(X_0, X_0), f(X_0, X_0)), f(f(X_0, X_0), f(X_0, X_0)))})$
$\quad = (true, \{\mathtt{X_3} \mapsto \mathtt{f(f(f(X_0, X_0), f(X_0, X_0)), f(f(X_0, X_0), f(X_0, X_0)))}\})$
$= (true, \{\ldots\})$

Folgerung: Der klassische Unifikationsalgorithmus benötigt im schlimmsten Fall **exponentiell viel Platz** (da die Terme nach der Unifikation exponentiell groß werden können) und hat eine **exponentielle Laufzeit** (occur check in exponentiell anwachsenden Termen).

Ursache: Baumdarstellung der Terme, wobei viele Unterbäume identisch sind.

Verbesserung des Algorithmus' durch eine andere Datenstruktur: Graphen statt Bäume.



Dadurch wird der Platzbedarf drastisch reduziert.

Im folgenden betrachten wir endliche, gerichtete, azyklische Graphen mit folgenden Eigenschaften:

- Die Knoten sind mit Variablen oder Funktoren markiert.
- Mit Variablen markierte Knoten haben keinen Nachfolger.

- Für eine Variable X gibt es höchstens einen mit X markierten Knoten.
- Ein mit einem Funktor f markierter Knoten hat genau n Nachfolger, wenn $f/n \in Func$.
- Zwischen zwei Knoten kann es mehr als eine Kante geben (Multigraph).
- Die von einem Knoten ausgehenden Kanten sind geordnet (d.h. es gibt einen 1., einen 2. usw. Nachfolger eines Knotens). $succ(G, v, k)$ liefert den k-ten Nachfolger des Knotens v im Graph G.

Solche Graphen nennen wir **Term-DAGs** ($TDAG$).

Wichtige Funktionen mit Term-DAGs:
term: $TDAG \times Node \to Term$ liefert den zu einem Knoten in einem Term-DAG zugehörigen Term:

$term(G, v) =$ IF $<v$ ist mit Variable X markiert>
THEN $return(X)$
ELSE $<v$ ist mit f markiert und hat n Nachfolger>
<sei $v_i = succ(G, v, i)$ für $i = 1, \ldots, n>$
$return(f(term(G, v_1), \ldots, term(G, v_n)))$
FI

replace: $TDAG \times Node \times Node \to TDAG$
$replace(G, v, w)$ (Voraussetzung: v und w sind Knoten in G, und es existiert kein Weg in G von w nach v) liefert einen Graphen G' mit:

1. G' hat die gleichen Knoten und Markierungen wie G
2. G' hat die gleichen Kanten wie G bis auf die Kanten, die in v einmünden: sie werden in G' ersetzt durch Kanten, die in w einmünden.

Das folgende Lemma setzt *replace* mit Variablensubstitutionen in Verbindung:

Lemma: Sei G ein Term-DAG, v ein Knoten in G, der mit der Variablen X markiert ist und σ die Substitution $\sigma = \{X \mapsto term(G, w)\}$ (w ist ebenfalls ein Knoten in G). Dann gilt für alle Knoten v' in G:

$$term(replace(G, v, w), v') = \text{IF } v' = v \text{ THEN } X \text{ ELSE } \sigma(term(G, v')) \text{ FI}$$

Im Graph $replace(G, v, w)$ ist v ein isolierter Knoten und dieser Graph repräsentiert den Term, der sich nach Anwendung einer Variablensubstitution ergibt (falls v mit einer Variablen markiert ist).

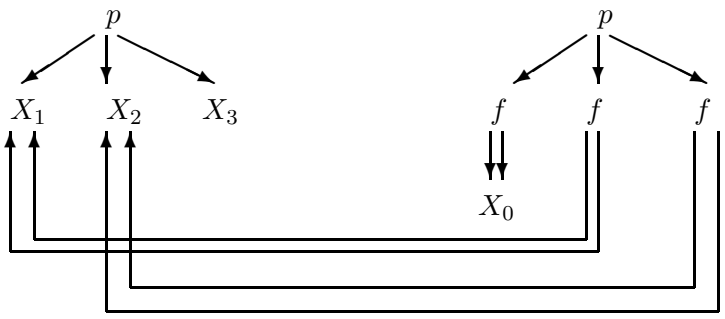
Verbesserter Algorithmus von Robinson [BC83]:

Gegeben: Term-DAG-Repräsentation G der zu unifizierenden Terme
Prozedur: **unify1**(v_1, v_2)
Eingabe: Knoten v_1 und v_2 in G
Ausgabe: $(bool, \sigma)$, wobei $bool = true$ g.d.w. $term(G, v_1)$ und $term(G, v_2)$ unifizierbar sind.
Falls $bool = true$, dann ist σ ein mgu (Seiteneffekt: Veränderung von G)

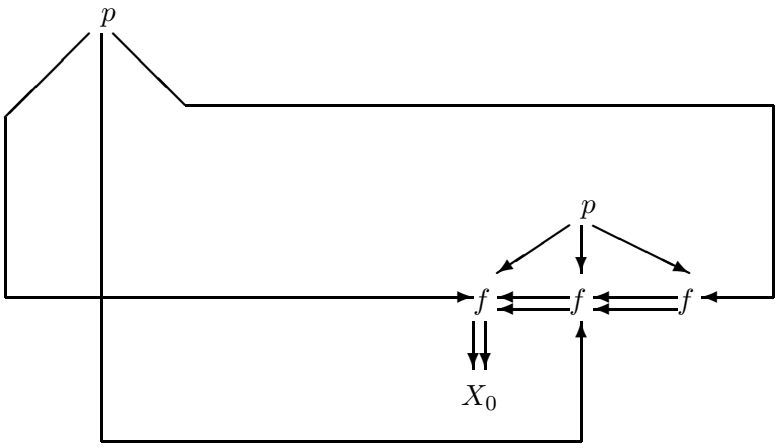
```
BEGIN
  IF v1 = v2 THEN bool := true; σ := {}
  ELSE  IF < v1 oder v2 ist mit einer Variablen markiert>
        THEN <Sei v mit einer Variablen markiert und w der andere Knoten>
             IF occur1(v, w)
             THEN bool := false
             ELSE  bool := true;
                   σ := {term(G, v) ↦ term(G, w)};
                   G := replace(G, v, w)
             FI
        ELSE  IF <v1 und v2 haben unterschiedliche Markierungen
                    oder eine unterschiedliche Anzahl von Nachfolgern>
             THEN bool := false
             ELSE  <Sei n die Anzahl der Nachfolger von v1 >
                   k := 0; bool := true; σ := {}
                   WHILE k < n AND bool
                   DO k := k + 1;
                       w1 := succ(G, v1, k);
                       w2 := succ(G, v2, k);
                       (bool, σ') := unify1(w1, w2);
                       IF bool THEN σ := σ' ∘ σ
                       FI
                   OD
             FI
        FI
  FI
  return(bool, σ)
END
```

Anmerkung: ***occur1(v, w)*** ist genau dann wahr, wenn es einen Weg in G von w nach v gibt (Vorkommenstest im Graphen). Zum effizienten Suchen eines Weges in einem Term-DAG sollte man die schon besuchten Knoten markieren. Das Rückgängigmachen der Markierungen ist nicht nötig, wenn jedesmal andere Markierungen benutzt werden (z.B. natürliche Zahlen).

Beispiel: Unifiziere $p(X_1, X_2, X_3)$ und $p(f(X_0, X_0), f(X_1, X_1), f(X_2, X_2))$.
Graph vor Beginn der Unifikation:



Graph nach erfolgreicher Unifikation (ohne isolierte Knoten):



Komplexität des Algorithmus' (ohne Berücksichtigung des Ausrechnens und der Ausgabe des mgu):

Speicherplatz: Die Anzahl der Knoten und Kanten wird nicht verändert. Daher ist die Speicherplatzkomplexität **linear** zur Größe der eingegebenen Term-DAGs.

Laufzeit: Da die Graphen nicht anwachsen, benötigt jeder Aufruf von *occur*1 (bei Verwendung von Markierungen) eine Laufzeit, die linear zur Größe der Terme ist. Bei jedem *replace*-Aufruf wird eine Variable eliminiert: Somit ist die Gesamtlaufzeit für *occur*1 und *replace*:

$$O(n \cdot p)$$

(n Anzahl der verschiedenen Variablen in den Termen, p Anzahl der Symbole in den Termen). Da aber ein kleiner Term-DAG einen exponentiell großen Term darstellen kann, kann die Anzahl der $unify1$-Aufrufe im schlechtesten Fall auch exponentiell sein, d.h. im schlechtesten Fall ergibt sich eine exponentielle Laufzeit.

Beispiel: Unifiziere

$$f(h(X_1, X_1), \ldots, h(X_{n-1}, X_{n-1}), Y_2, \ldots, Y_n, Y_n)$$

mit

$$f(X_2, \ldots, X_n, h(Y_1, Y_1), \ldots, h(Y_{n-1}, Y_{n-1}), X_n)$$

Bei der Unifikation dieser Terme wachsen zunächst die X_i und dann die Y_j zu exponentieller Größe an. Die Unifikation des letzten Argumentpaares benötigt daher exponentiell viele $unify1$-Aufrufe.

Die Anzahl der $unify1$-Aufrufe kann beschränkt werden, wenn für strukturell gleiche Terme der Aufruf von $unify1$ nicht ausgeführt wird. Um die strukturelle Gleichheit schnell zu testen, werden strukturell gleiche Terme, die ja nach einer erfolgreichen Unifikation entstehen, zu einem Term zusammengefaßt. Dazu wird die *replace*-Funktion verwendet, die nun nicht nur auf Variablenknoten, sondern auch auf Funktorknoten angewendet wird. Mit dieser Idee erhalten wir den effizienten Algorithmus:

Effizienter Algorithmus von Robinson [BC83]:

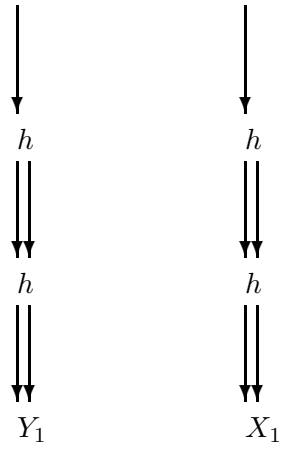
Gegeben: Term-DAG-Repräsentation G der zu unifizierenden Terme
Prozedur: **unify2**(v_1, v_2)
Eingabe: Zwei *verschiedene* Knoten v_1 und v_2 in G
Ausgabe: $(bool, \sigma)$, wobei $bool = true$ g.d.w. $term(G, v_1)$ und $term(G, v_2)$ unifizierbar sind.
Falls $bool = true$, dann ist σ ein mgu.
(Seiteneffekt: Alle in v_1 oder v_2 einmündenden Kanten
münden danach in einen der beiden Knoten)

```
BEGIN
  IF < v1 oder v2 ist mit einer Variablen markiert>
  THEN < Sei v mit einer Variablen markiert und w der andere Knoten>
       IF occur1(v, w)
       THEN bool := false
       ELSE bool := true;
            σ := {term(G, v) ↦ term(G, w)};
            G := replace(G, v, w)
       FI
  ELSE IF < v1 und v2 haben unterschiedliche Markierungen oder
            eine verschiedene Anzahl von Nachfolgern>
       THEN bool := false
       ELSE < Sei n die Anzahl der Nachfolger von v1 >
            k := 0; bool := true; σ := {};
            WHILE k < n AND bool
            DO k := k + 1;
               w1 := succ(G, v1, k);
               w2 := succ(G, v2, k);
               IF w1 ≠ w2
               THEN (bool, σ') := unify2(w1, w2);
                    IF bool THEN σ := σ' ∘ σ
                    FI
               FI
```

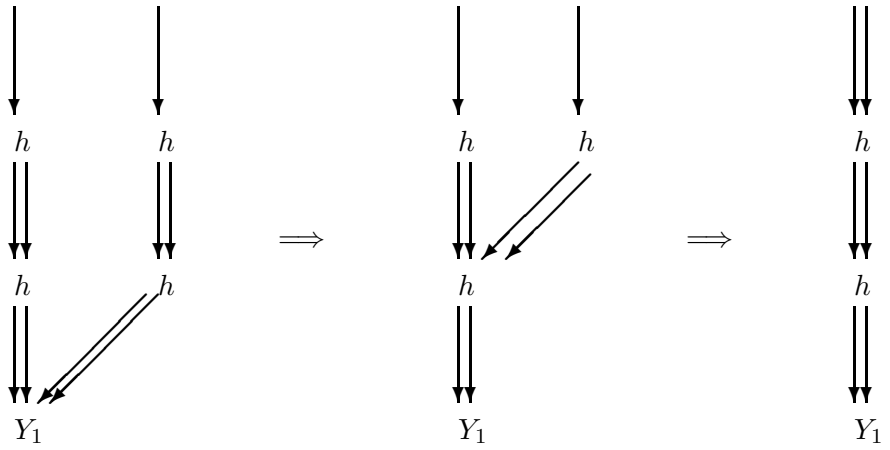
OD
IF *bool* THEN $G := replace(G, v_1, v_2)$
FI
FI
FI
$return(bool, \sigma)$
END

Beispiel: Unifikation der Terme im letzten Beispiel.

Wenn die letzten Argumente (Y_n und X_n) unifiziert werden müssen, bedeutet dies aufgrund der bisherigen Substitutionen, daß die Term-DAGs



(für $n = 3$) unifiziert werden müssen. Dabei entstehen bei $unify2$ die folgenden Term-DAGs (isolierte Knoten werden weggelassen):



Implementierungshinweise:

- G ist eine globale Datenstruktur.

- Der „occur check“ kann durch Verwendung von Knotenmarkierungen effizient realisiert werden.
- *replace* kann effizient durch Rückwärtszeiger implementiert werden.
- Falls der mgu explizit ausgegeben werden soll: Führe für jede Variable einen Extra-Knoten ein, von dem am Anfang eine Kante zum Variablenknoten geht. Am Ende geht von diesem Knoten eine Kante zu dem Term (dargestellt als Term-DAG), durch den die Variable ersetzt werden muß.

Laufzeitabschätzung (ohne mgu-Ausgabe und -Berechnung):
In jedem $unify2$-Aufruf wird der Graph um einen Knoten verkleinert, d.h. die Anzahl der $unify2$-Aufrufe ist höchstens so groß wie die Anzahl der Knoten im Anfangsgraphen. Da $occur1$ eine lineare Laufzeit hat, hat die letzte Version des Unifikationsalgorithmus eine „worst-case“-Laufzeit von

$$O(p^2)$$

wenn p die Anzahl der Symbole in den zu unifizierenden Termen ist. Die explizite Ausgabe des mgu (in Termform und nicht in Graphendarstellung) kann jedoch exponentiell viel Speicherplatz und Laufzeit beanspruchen.

Anmerkungen zur Unifikation:

- Die Effizienz von Unifikationsalgorithmen ist stark abhängig von der gewählten Datenstruktur.
- Paterson und Wegman [PW78] haben einen linearen Unifikationsalgorithmus gefunden, der auf der Einteilung von Knoten in Äquivalenzklassen basiert und eine geschickte Datenstruktur zur Implementierung benutzt. Der Unifikator wird nicht explizit, sondern implizit als Folge

  $$[x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n]$$

  dargestellt. Diese ist sequentiell zu lesen: Zunächst wird die Variable x_1 durch den Term t_1 ersetzt, dann die Variable x_2 durch t_2 (wobei Vorkommen von x_2 in t_1 auch ersetzt werden müssen) usw. Hierdurch wird die exponentielle Größe der dargestellten Terme vermieden.
- Martelli und Montanari [MM82] haben ebenfalls einen effizienten Unifikationsalgorithmus angegeben, der auf der Umformung und Lösung von Gleichungssystemen basiert (Unifikation der Argumente zweier Terme bedeutet Übergang von der Gleichung $f(r_1, \ldots, r_n) = f(s_1, \ldots, s_n)$ zu dem Gleichungssystem $r_1 = s_1, \ldots, r_n = s_n$).
- In fast allen heutigen Prolog-Systemen wird jedoch der klassische Algorithmus verwendet (genauer: eine Mischung aus dem klassischen Algorithmus und der ersten Verbesserung), weil er die einfachste Datenstruktur hat und exponentiell wachsende Terme in der Praxis selten vorkommen. Dafür wird allerdings aus Effizienzgründen häufig auf den Vorkommenstest verzichtet, was aus theoretischer Sicht eine Katastrophe ist. Um Speicherplatz zu sparen, werden verschiedene Techniken angewendet, die wir im nächsten Abschnitt besprechen.

2.4 Speicherverwaltung

Literatur: [Bru82] [Mel82] [vE84] [Bru84] [War84]

Die prinzipielle Arbeitsweise eines Prolog-Interpreters ist in Abschnitt 2.2 dargestellt worden. Welche Datenstrukturen sind zur effizienten Implementierung eines solchen Interpreters zu wählen?

Wichtigste Operationen des Interpreters:

- Unifikation eines Literals mit einem Klauselkopf
- Beweisen („Aufrufen“) des Klauselrumpfes
- Backtracking: Rücksetzen auf den letzten Berechnungszustand, Ausprobieren neuer Alternativen

Im letzten Interpreter mit „structure sharing“ wurde zu jedem Teilziel die bisher aufgebaute Substitution mitgeführt. Dadurch ist das Backtracking einfach (Löschen des gerade zu beweisenden Ziels), aber dieser Interpreter erfordert viel Speicherplatz, da die einzelnen Substitutionen im Verlauf der Berechnung groß werden können. In diesem Abschnitt werden wir daher bessere Speichertechniken angeben.

Grundlegende Idee des „structure sharing“-Interpeters:

Die Programmklauseln (dies entspricht dem Programmcode in herkömmlichen Programmiersprachen) werden bei jeder Anwendung nicht (unter Berücksichtigung des Unifikators) kopiert, sondern die aktuelle Instanz einer Klausel wird durch einen Verweis auf die Klausel und die aktuelle Belegung der Variablen in der Klausel dargestellt. Wie bei imperativen Programmiersprachen wird auch hier zur Laufzeit unterschieden zwischen einem fixen Anteil (Programmcode) und einem variablen Anteil (Umgebung, Bindung der Variablen).

Beispiel: Zu beweisendes Literal: `p(g(a),B,f(C,b))`

Zum Beweis verwendete Klausel: `p(X,Y,f(X,Z))` $\leftarrow \cdots$

Darstellung der Klausel zur Laufzeit:

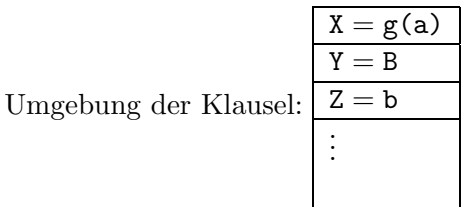
`p(X,Y,f(X,Z))` $\leftarrow \cdots$

Umgebung der Klausel:

X
Y
Z
⋮

Darstellung der Klauselinstanz nach Unifikation:

`p(X,Y,f(X,Z))` $\leftarrow \cdots$



Dies entspricht der Klausel `p(g(a),B,f(g(a),b))` $\leftarrow \cdots$

Jedes zu beweisende Ziel kann also dargestellt werden als ein Paar bestehend aus einem Zeiger in eine Programmklausel und einer Umgebung. Somit ist die Hauptdatenstruktur des Interpreters ein Stack, der Umgebungen und Zeiger in Programmklauseln enthält. Daher nennen wir ihn **Umgebungsstack**. Außerdem benötigt man noch folgende Informationen:

- Verweis auf Teilziele, die noch bewiesen werden müssen, wenn das aktuelle Teilziel bewiesen worden ist (dies ist die Liste *RenamedGoal* im obigen Interpreter und kann durch einen Verweis auf den „Vaterknoten“ im Beweisbaum realisiert werden)
- Verweis auf alternative, noch nicht benutzte Klauseln (die dann bei einem Fehlschlag ausprobiert werden können)
- Liste der Variablen, die durch Unifikation gebunden worden sind (diese müssen beim Backtracking wieder als ungebunden markiert werden)

Folgende Register beschreiben also den aktuellen Zustand einer Berechnung:

CURR-CALL: Verweis auf den Programmcode des aktuellen Teilziels

CURR-ENV: Verweis auf die Umgebung, die zu dem Teilziel CURR-CALL gehört, d.h. Verweis auf die Umgebung der Klausel, in deren Rumpf CURR-CALL liegt

CURR-CLAUSE: Verweis auf die Klausel, die im nächsten Schritt zum Beweis des ersten Literals von (CURR-CALL, CURR-ENV) angewendet wird

LASTBACK: Verweis auf den letzten Backtrackpunkt, in dem die Information zum Backtracking gespeichert ist (ein Backtrackpunkt ist eine Umgebung mit zusätzlichen Informationen)

Somit sind CURR-CALL und CURR-CLAUSE Zeiger in das Prolog-Programm, während CURR-ENV und LASTBACK in den Umgebungsstack verweisen. Der Umgebungsstack enthält Umgebungen und Verwaltungsinformationen (Backtrackpunkte). Für die Liste der beim Backtracking zurückzusetzenden Variablen wird in der Regel noch ein eigener Stack (trail) verwendet.

Zur besseren Speicherausnutzung unterscheiden wir zwischen **deterministischen** und **nichtdeterministischen Aufrufen**. Ein Aufruf ist deterministisch, wenn es nach der aktuell angewendeten Klausel keine weitere passende Klausel gibt.

Für jeden deterministischen Aufruf eines Literals enthält der Umgebungsstack folgende Informationen:

CALL: Ein Verweis auf den Programmcode des Vaters des aufgerufenen Literals

FATHER: Verweis auf die Umgebung, die zu dem aufgerufenen Literal gehört, d.h. Verweis auf die Umgebung der Klausel, aus deren Rumpf das aufgerufene Literal stammt

BINDINGS: Eine Liste der Variablenbindungen für die Variablen, die in der gerade angewendeten Klausel vorkommen

Ein nichtdeterministischer Aufruf enthält außerdem:

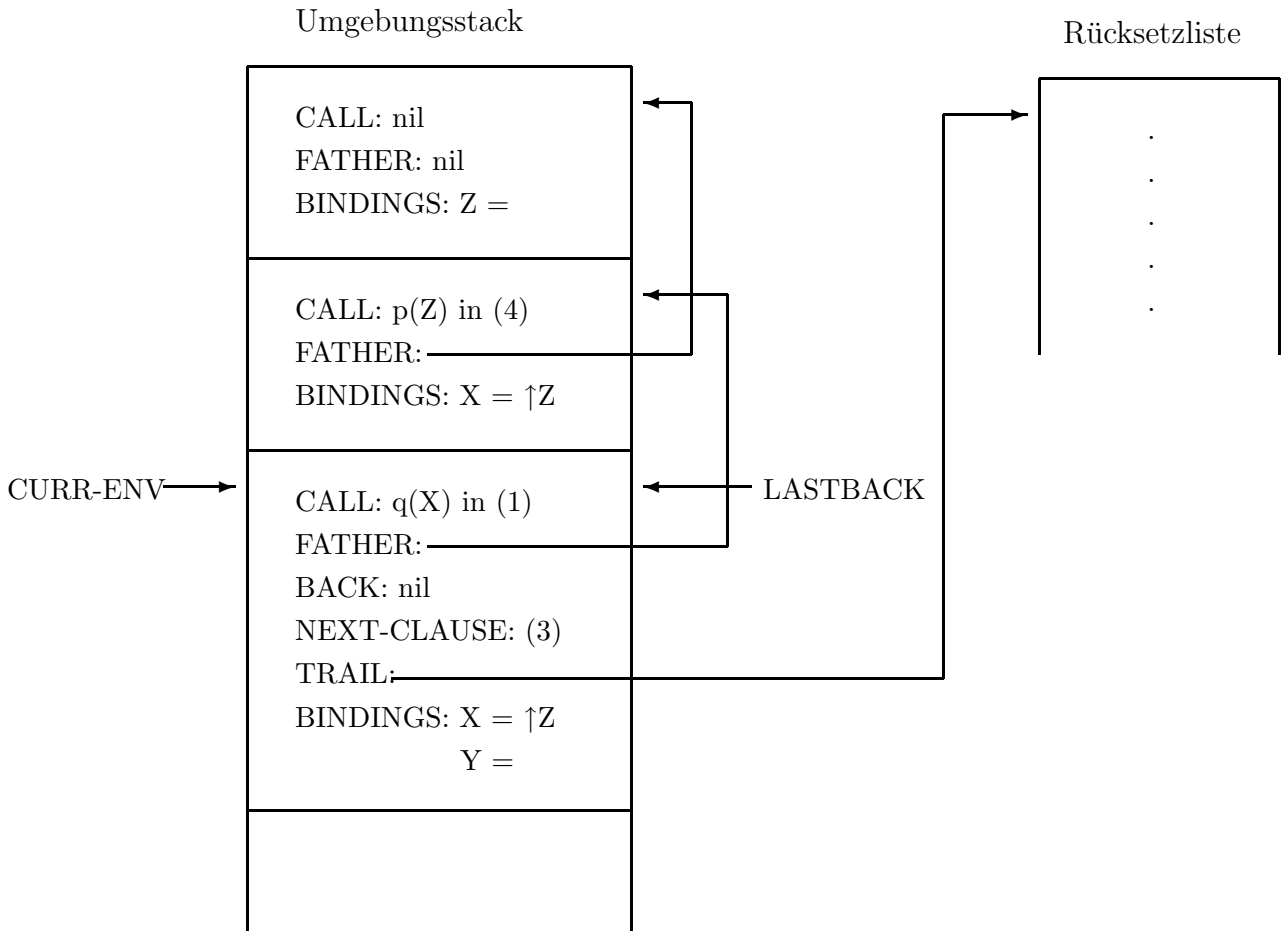
BACK: Ein Verweis auf den vorhergehenden Backtrackpunkt

NEXT-CLAUSE: Verweis auf die Klausel, die beim nächsten Beweisversuch für das Literal angewendet wird

TRAIL: Verweis auf die Liste der zurückzusetzenden Variablen

Beispiel: Programm: **(1)** `p(X)` ← `q(X)`
(2) `q(X)` ← `r(X,Y), p(Y)`
(3) `q(X)` ← `s(X,Y)`
Anfrage: **(4)** `p(Z)`
Abarbeitung: `p(Z)`
→ `q(Z)`
→ `r(Z,Y), p(Y)`

Zustand des Interpreterspeichers an dieser Stelle:



CURR-CALL: r(X,Y) in (2)
CURR-CLAUSE: 1. Klausel für r/2

Interpreter mit verbesserten Datenstrukturen:

Eingabe: Programm P, Ziel G

Ausgabe: Substitution für die Variablen in G

BEGIN

Initialisiere die Datenstrukturen wie folgt:

Umgebungsstack

CURR-ENV→ | CALL: nil
FATHER: nil
BINDINGS: <alle in G vorkommenden Variablen> |

CURR-CALL: <1. Literal in G>

CURR-CLAUSE: <1. Klausel in P, die zu CURR-CALL paßt>

LASTBACK: nil

```
stop := false;
WHILE not stop
DO <Erzeuge neuen Eintrag new-entry auf Umgebungsstack mit:>
    CALL := CURR-CALL;
    FATHER := CURR-ENV;
    next := <1. Nachfolger von CURR-CLAUSE in P, der zu CURR-CALL paßt>
    IF next = nil
    THEN <markiere neuen Eintrag als deterministisch>
    ELSE  <markiere neuen Eintrag als nichtdeterministisch>
          BACK := LASTBACK;
          LASTBACK := <Verweis auf new-entry>;
          NEXT-CLAUSE := next;
          TRAIL := top of TRAIL
    FI
    BINDINGS := <alle in CURR-CLAUSE vorkommenden Variablen>;
    <Unifiziere CURR-CALL (Bindungsumgebung: CURR-ENV) mit Kopf der Klausel
      CURR-CLAUSE (Bindungsumgebung: new-entry); alle dabei entstehenden Bindungen
      für Variablen außerhalb von new-entry werden auf TRAIL notiert>;
    IF <Unifikation erfolgreich>
    THEN {Finde nächstes, zu beweisendes Literal}
          CURR-CALL := <1. Literal im Rumpf von CURR-CLAUSE oder nil, falls
                        der Rumpf leer ist>;
          CURR-ENV := new-entry;
          WHILE not stop AND CURR-CALL = nil
          DO CURR-CALL := <Nachfolger von CALL in CURR-ENV>;
              CURR-ENV := FATHER of CURR-ENV;
              stop := (FATHER of CURR-ENV) = nil AND CURR-CALL = nil
          OD
          IF stop
          THEN <gebe Bindungen in CURR-ENV aus>
          ELSE  CURR-CLAUSE := <1. Klausel in P, die zu CURR-CALL paßt>
          FI
    ELSE  {Backtracking-Schritt}
          IF LASTBACK = nil
          THEN stop := true
          ELSE  <Setze alle Variablen auf „ungebunden“, die in der Rücksetzliste oberhalb
                  von TRAIL of LASTBACK notiert sind und lösche diese Einträge in der
                  Rücksetzliste>;
                CURR-CALL := CALL of LASTBACK;
                CURR-ENV := FATHER of LASTBACK;
                CURR-CLAUSE := NEXT-CLAUSE of LASTBACK;
                new-back := BACK of LASTBACK;
                <Lösche alle Einträge von Top bis einschließlich LASTBACK>;
```

```
                    LASTBACK := new-back
                FI
            FI
        OD
END
```

Anmerkungen:

- In diesem Interpreter muß man zur Verwaltung des Backtracking „die nächste Klausel, die zu einem Literal paßt“ finden. Im einfachsten Fall ist dies die nächste Klausel, deren Kopf den gleichen Prädikatbezeichner wie das Literal hat. Dazu kann man z.B. alle Klauseln für ein Prädikat miteinander verketten. Effizienter ist es, zusätzlich über die Argumente der Klauselköpfe zu indizieren. Diese Technik wird in übersetzenden Prolog-Systemen angewendet. Wir werden später darauf zurückkommen.
- Wenn eine Lösung gefunden ist, d.h. nach Ausgabe der Variablenbindungen, stoppt der Interpreter. Falls gewünscht, kann in diesem Fall mit dem Backtracking-Schritt weitergemacht werden, um weitere Lösungen zu finden.
- An diesem Interpreter sind noch zahlreiche Optimierungen möglich, von denen jetzt einige vorgestellt werden.

Optimierungen des Interpreters:

1. *Löschen von deterministischen Aufrufen:* Die Informationen von deterministischen Aufrufen werden nicht mehr benötigt, wenn diese komplett bewiesen sind, d.h. wenn
   - CURR-ENV ein deterministischer Aufruf ist und ganz oben auf dem Umgebungsstack liegt
   - CURR-CALL = nil

   dann kann dieser Aufruf vor weiteren Aktionen gelöscht werden, d.h. führe die Aktionen

   CURR-CALL:=<Nachfolger von CALL in CURR-ENV>;
   CURR-ENV:= FATHER of CURR-ENV;
   <Lösche obersten Eintrag im Umgebungsstack>;

   aus. Voraussetzung hierfür: Alle Variablenbindungen sind im Umgebungsstack von oben nach unten orientiert, d.h. es gibt keine Zeiger in Richtung TOP (sonst verweisen einige Zeiger in undefinierte Umgebungen). Die Unifikationsprozedur ist entsprechend zu modifizieren.

2. *Sortieren von Variablen in Umgebungen:* Wenn der Rumpf der Klausel

   `q(X)` ← `r(X,Y), p(Y)`

bewiesen wird, dann enthält die Umgebung die Bindungen der Variablen `X` und `Y`. Sobald aber `r(X,Y)` komplett bewiesen ist, wird nur noch die Bindung von `Y` benötigt, und die Bindung von `X` kann in dieser Umgebung gelöscht werden. Diese Technik („environment trimming“) wird in Prolog-Compilern eingesetzt und wir werden später darauf zurückkommen.

3. Tail recursion optimization (TRO):

   Klausel: `p(...)` ← `..., q(...)`

   wenn vor dem Beweis des letzten `q`-Literals kein Backtrackpunkt oberhalb der Umgebung für diese Klausel auf dem Umgebungsstack liegt und das letzte `q`-Literal nur mit *einem* Klauselkopf unifizierbar ist, dann kann die Umgebung für diese Klausel direkt nach der Unifikation von `q(...)` mit dem Klauselkopf gelöscht werden. Dadurch kann sehr viel Speicherplatz eingespart werden (z.B. `append`). Vorsicht: Noch vorhandene Variablenbindungen in der Umgebung! Genaueres: Kapitel 4 über Prolog-Übersetzer oder [Bru82].

Termrepräsentation

Im bisherigen Interpreter sind Terme immer dargestellt als Paar (Skelett, Bindungen). Dabei ist „Skelett“ ein Teil des Programmcodes, während „Bindungen“ eine Liste der aktuellen Bindungen der im Skelett vorkommenden Variablen ist.

Beispiel: Das Paar (f(X, Y, X), {X = a, Y = g(b)}) stellt den Term f(a, g(b), a) dar. In der Implementierung werden die Namen der Variablen nicht gespeichert, sondern Variablen werden durch ihren „offset“ in der Umgebung repräsentiert:

(f(VAR(0), VAR(1), VAR(0)), {0:a, 1:g(b)})

Im Interpreter werden alle Umgebungen (Bindungslisten) auf dem Stack gespeichert. Bei der **Unifikation** haben wir daher drei Fälle beim Binden von Variablen zu unterscheiden:

1. Eine Variable X, die in der Umgebung E_i ungebunden ist, wird mit einer Variablen Y, die in Umgebung E_j ungebunden ist, unifiziert. Annahme: E_j ist jünger als E_i oder gleich (d.h. E_j liegt auf dem Stack nicht unterhalb von E_i). Dann wird Y in E_j an X in der Umgebung E_i gebunden. Der so entstandene neue Zeiger von E_j nach E_i verweist also in die untere Richtung des Umgebungstacks.

2. Eine Variable X, die in der Umgebung E_i ungebunden ist, wird mit einem Term t mit der Umgebung E_j unifiziert und E_i ist jünger als E_j. Dann wird X in E_i an den Term t mit der Umgebung E_j gebunden. Der so entstandene Zeiger von E_i nach E_j verweist wiederum in die untere Richtung des Umgebungsstacks.

3. Eine Variable X, die in der Umgebung E_i ungebunden ist, wird mit einem Term t mit der Umgebung E_j unifiziert und E_j ist jünger als E_i. Dann muß X in E_i an den Term t mit der Umgebung E_j gebunden werden. Der so entstandene Zeiger von E_i nach E_j verweist aber in die obere Richtung des Stacks, d.h. wichtige Speicheroptimierungen im Interpreter (Löschen von deterministischen Aufrufen, TRO) sind nicht anwendbar!

Beispiel:

CURR-CALL:	$p(R, S, T)$
CURR-ENV:	$R:$ $(\uparrow f(\ldots), E_R)$
	$S:$ ungebunden
	$T:$ ungebunden
CURR-CLAUSE:	$p(X, Y, g(X, Z)) \leftarrow \ldots$

Der Interpreter legt zunächst eine neue Umgebung E_n für X, Y, Z auf dem Stack an. Anschließend erfolgt die Unifikation:

- Binde X in E_n an $(\uparrow f(\ldots), E_R)$ (Fall 2)
- Binde Y in E_n an S in CURR-ENV (Fall 1)
- Binde T in CURR-ENV an $(\uparrow g(X, Z), E_n)$ (Fall 3)

(So wurde es z.B. im Interpreter von [BM72] gemacht.)

Gesucht: Lösung, bei der kein Zeiger im Stack von einer älteren in eine neuere Umgebung zeigt.

Idee [War77]: Führe einen zweiten Stack ein („globaler Stack“), in dem die problematischen Terme (Umgebungen) gespeichert werden. Problematische Terme sind solche, bei dem während der Unifikation evtl. Zeiger von älteren Umgebungen in neuere Umgebungen gesetzt werden müssen, wie z.B. $g(X, Z)$ im letzten Beispiel. Unterscheide zu diesem Zweck zwischen **lokalen** und **globalen** Variablen. Die Bindungen aller lokalen Variablen werden im Umgebungsstack (oder auch „local stack“) gespeichert, die der globalen Variablen im globalen Stack. Der Umgebungsstack wird wie bisher verwaltet (Optimierungen sind möglich, da alle Zeiger nach unten zeigen), der globale Stack wird nur beim Backtracking verkleinert. Alle Zeiger im Umgebungsstack verweisen nach unten in den Umgebungsstack oder in den globalen Stack.

Im obigen Beispiel sind X und Z globale Variablen, und daher liegt die neue Bindungsumgebung E_{XZ} für X und Z auf dem globalen Stack. Somit wird T in CURR-ENV an $(\uparrow g(X, Z), E_{XZ})$ gebunden, was unproblematisch ist.

Somit wird jede **Klauselinstanz** (und jeder Term) durch einen Verweis in den Programmcode und eine lokale und eine globale Umgebung repräsentiert. Der globale Stack ist eine Liste von Variablenbindungen, während der lokale Stack zusätzlich Verwaltungsinformationen enthält.

Die notwendigen Änderungen am Interpreter dürften nach diesen Ausführungen klar sein. Wir geben nun nur noch an, wie man die Variablen in einer Klausel klassifizieren kann:

Variablenklassifikation beim Interpreter mit „structure sharing“ (nach [War77]):

Typ	Kriterium	Kommentar
global	Kommt in mindestens einer Struktur vor	Bindungen werden auf dem globalen Stack gespeichert. Lebenszeit endet erst beim Backtracking.
lokal	Kommt mehrfach vor, wobei mindestens einmal im Rumpf und keinmal in einer Struktur	Bindungen werden auf dem lokalen Stack gespeichert. Lebenszeit endet, wenn die Klausel vollständig abgearbeitet ist (also auch am Ende von deterministischen Klauseln).
temporär	Kommt mehrfach vor, aber nur im Kopf und keinmal in einer Struktur	Bindungen werden in einem Maschinenregister gespeichert. Wird nur während der Unifikation benötigt.
nichtig	Kommt nur einmal vor	Braucht nicht gespeichert zu werden.

Diese feinere Klassifizierung spart Speicherplatz (temporäre und nichtige Variablen werden auf den Stacks nicht gespeichert). Wenn spezialisierte Unifikationsinstruktionen generiert werden (vgl. Kapitel 4 über die Übersetzung von Prolog-Programmen), dann entsteht auch kein Overhead durch diese zusätzliche Klassifikation.

Vorteile des „structure sharing“:

- Geringer Speicherplatzverbrauch
- Kein Kopieren von Termen, nur Anlegen von Umgebungen

Nachteile des „structure sharing“:

- Variable besteht aus zwei Zeigern (auf Programmcode + Umgebung)
- Unifikation ist aufwendiger (da die Termstruktur aus dem Skelett und den Bindungen konstruiert werden muß)
- Es können lange Verweisketten entstehen, ein Term kann sehr verstreut im Speicher sein (Problem bei Paging-Systemen)

Alternative zu „structure sharing“: „structure copying“:

- Die Terme werden „direkt“ repräsentiert, d.h. Term der Form $f(t_1, \ldots, t_n)$ wird in $n+1$ aufeinanderfolgenden Speicherzellen in der Form $f(\uparrow t_1, \ldots, \uparrow t_n)$ gespeichert. Die Argumente sind entweder direkt gespeichert (bei ungebundenen Variablen oder Konstanten) oder enthalten Verweise auf andere komplexe Terme.
- Es gibt keine Verweise von Variablenbindungen in den Programmcode. Wenn eine solche Bindung notwendig ist, wird der betreffende Teilterm aus dem Code in den globalen Stack *kopiert*.
- Wenn eine ungebundene Variable mit einem Term im globalen Stack unifiziert wird, wird diese an den Term gebunden (Zeiger auf den Term).

- Wenn eine ungebundene Variable mit einem Term im Programmcode unifiziert wird, wird dieser Term auf den globalen Stack kopiert. Dieser Term kann Variablen enthalten, deren aktueller Wert aus der zugehörigen Umgebung hervorgeht. Falls diese Variable in der Umgebung
  - ungebunden ist, dann wird im kopierten Term eine ungebundene Variable angelegt,
  - an einen Term (im globalen Stack) gebunden ist, dann wird im kopierten Term anstelle der Variablen ein Verweis auf diesen Term angelegt.
- Der globale Stack heißt wegen seiner geänderten Funktion auch „copy stack“ oder „heap“. Er wird aber ebenfalls nur beim Backtracking verkleinert.

Beispiel:

CURR-CALL:	$p(R, S, T)$
CURR-ENV:	$R:$ $\uparrow f(\ldots)$ (Zeiger auf die direkte Darstellung des Terms $f(\ldots)$)
	$S:$ ungebunden
	$T:$ ungebunden
CURR-CLAUSE:	$p(X, Y, g(X, Z)) \leftarrow \ldots$

Bei der Unifikation werden X, Y, Z in ihrer neuen Umgebung E_n wie folgt gebunden:

- X in E_n verweist auf die schon existierende Darstellung des Terms $f(\ldots)$
- Y in E_n verweist auf S in CURR-ENV
- T in CURR-ENV verweist auf eine neu angelegte Kopie von $g(X, Z)$. Dabei wird das erste Argument in der Kopie ein Verweis auf die Darstellung des Terms $f(\ldots)$, und das zweite Argument ist eine neue freie Variable Z'
- Z in E_n wird an die neue Variable Z' gebunden

Vergleich „structure sharing“ - „structure copying“:

„structure sharing“	„structure copying“
Variablenbindung besteht aus zwei Zeigern	Variablenbindung besteht aus einem Zeiger
Globaler Stack enthält nur Bindungen	Copy stack enthält Terme (+ Bindungen)
Terme können stark gestreut sein (Problem bei Paging-System)	Kompaktere Darstellung der Terme
Lange Verweisketten möglich (Nachteil bei Selektion)	Kurze Verweisketten, dafür neue Kopien bei Konstruktionen von Termen

Der Vergleich des Speicherverbrauchs ist schwierig, da je nach Anwendung „structure sharing“ oder „structure copying“ besser ist (vgl. [Mel82]). Von der Implementierung ist „structure copying“ etwas einfacher. C-Prolog [Per87] und der DEC-10-Compiler [War77] verwenden „structure sharing“. [War83] (und darauf basierende Compiler) verwendet „structure copying“ (und „structure sharing“ für Programmklauseln).

Anmerkung:

Da der globale Stack nur beim Backtracking verkleinert wird, entstehen auf diesem Stack häufig Datenstrukturen, die vom laufenden Programm nicht mehr referenziert werden. Daher ist die Implementierung eines „garbage collectors“ für den globalen Stack sinnvoll [Bru84].

Kapitel 3

Vom Interpreter zum Compiler

Literatur: [KC84], [Kur86]

Der Interpreter ist sehr allgemein gehalten, da er das auszuführende Programm nicht kennt. Wenn dieses Programm aber bekannt ist, können viele Dinge schon zur Übersetzungszeit erledigt werden, die sonst erst zur Laufzeit gemacht werden. Ziel ist es, den Interpreter für ein gegebenes Programm zu spezialisieren.

Einfaches Beispiel: Das Prädikat $p/2$ sei definiert durch die Klausel $p(X, X) \leftarrow$. Wenn der Interpreter das Literal $p(t_1, t_2)$ beweisen muß, führt er folgende Aktionen aus:

1. Heraussuchen aller zu p gehörigen Klauseln (d.h. $p(X, X) \leftarrow$).
2. Prüfen, ob p deterministisch ist (ja, da es nur eine Klausel gibt).
3. Anlegen einer neuen Umgebung mit Variable X.
4. Unifikation von $p(t_1, t_2)$ und $p(X, X)$.

Wenn aber zur Übersetzungszeit bekannt ist, daß $p/2$ nur durch die Klausel $p(X, X) \leftarrow$ definiert ist, kann ein Beweis des Literals $p(t_1, t_2)$ in folgende Laufzeitaktion übersetzt werden:

Unifiziere t_1 mit t_2.

Hier sieht man deutlich die Möglichkeiten, Interpreter zu spezialisieren. Dies ist genau das, was Compiler im wesentlichen machen. Ein Compiler für Prolog kann für das gegebene Programm folgende Spezialisierungen im Vergleich zum Interpreter vornehmen:

- Spezielle Unifikation für Klauselköpfe
- Anlegen von Umgebungen nur bei Bedarf
- Anlegen von Backtrackpunkten nur bei Bedarf (macht teilweise auch schon unser Interpreter), z.B. mit besonderem Indizierungsschema für Klauselköpfe
- „tail recursion optimization“

Insbesonders der ersten Optimierung werden wir uns in diesem Kapitel widmen. Die anderen Optimierungen werden im nächsten Kapitel besprochen.

3.1 Partielle Auswertung

Ein **partieller Auswerter** ist ein Programm, welches andere Programme in möglichst effizientere transformiert. Die Transformation ist meistens möglich aufgrund der Tatsache, daß Teile der Eingaben bekannt sind. In [JSS85] wird gezeigt, wie partielle Auswerter benutzt werden können, um aus Interpretern Compiler zu erzeugen:

- Ein *Interpreter* nimmt ein zu interpretierendes Programm und die Programmeingaben als Eingaben und arbeitet es entsprechend der Semantik ab.
- Ein *Compiler* nimmt ein Programm als Eingabe und produziert ein Programm, welches seinerseits die Programmeingaben einliest und sie entsprechend verarbeitet.
- Idee: Wenn man den Interpreter und das zu interpretierende Programm kennt, dann kann man dies partiell auswerten und erhält somit das übersetzte Programm.

Diese Idee werden wir an einem Beispiel erläutern, um damit die heute üblichen Übersetzungstechniken zu motivieren. Zunächst geben wir daher einen Überblick über partielle Auswertungstechniken für Prolog (Literatur: [Ven84] [TF86] [VD88] [FA88] [LS88] [SB89]).

Def.: Ein **partieller Auswerter** für Prolog nimmt ein Prolog-Programm als Eingabe und produziert ein semantisch äquivalentes Prolog-Programm.

Anmerkung: In der Regel ist es zulässig, daß

- das neue Programm nur semantisch äquivalent bzgl. bestimmter Anfragen sein muß, d.h. der partielle Auswerter hat auch noch bestimmte Anfragen als Eingabe,
- das neue Programm terminieren darf für Anfragen, bei denen das alte nicht terminiert.

Wichtige Transformationen:

- **„Entfalten“** (unfolding) von Literalen:

  Sei L Literal und $L_i \leftarrow G_i$ $(i = 1, \ldots, k)$ seien Varianten aller Klauseln, die zu L „passen“. Dann ersetze L durch $(G'_1; \ldots; G'_k)$ mit $G'_i = (t_{01} = t_{i1}, \ldots, t_{0n} = t_{in}, G_i)$, falls $L = p(t_{01}, \ldots, t_{0n})$ und $L_i = p(t_{i1}, \ldots, t_{in})$ $(i = 1, \ldots, k)$. (Dies ist sinnvoll, wenn man die Disjunktion durch Anwendung der anderen Transformationen eliminieren kann).

  Beispiel: Gegeben sei die folgende Standarddefinition zur Konkatenation von Listen:

  ```
  append([], L, L) ←
  append([E|R], L, [E|RL]) ← append(R, L, RL)
  ```

  Dann kann man `append([1], L2, L3)` ersetzen durch

  (1) `([] = [1], L = L2, L = L3; [E|R] = [1], L = L2, [E|RL] = L3, append(R, L, RL))`

- **„Vorwärtsunifikation“:** Bei expliziter Unifikation (Prädikat `=/2`) kann das Ergebnis, falls es teilweise berechenbar ist, vorwärts propagiert werden:

  Beispiel (1) wird ersetzt durch

  (2) `([] = [1], L = L2, L2 = L3; [E|R] = [1], L = L2, [1|RL] = L3, append([], L2, RL))`

- **„Rückwärtsunifikation“:** Analog zu Vorwärtsunifikation, jedoch keine Propagierung zu nichtlogischen Literalen.

  Beispiel: Die Ersetzung von `(X = Y, Y = a)` durch `(X = a, Y = a)` ist erlaubt, jedoch ist die Ersetzung von `(var(X), X = a)` durch `(var(a), X = a)` falsch!

- **Löschen** von Unifikation: Explizite Unifikation kann gelöscht werden, falls die beteiligten Variablen auf einer Seite nur einmal auftreten oder gar keine Variablen beteiligt sind.

  Beispiel: `[] = [1]` $\rightarrow$ `fail`
  `[E|R] = [1]` $\rightarrow$ `true` (falls `E, R` sonst nicht vorkommen)
  Somit wird aus (2):
  (3) `(fail, true, L2 = L3; true, true, [1|RL] = L3, append([], L2, RL))`

- **Ausrechnen** von Standardprädikaten:

  Beispiel: `(fail,` $L_1, \ldots, L_k$) $\rightarrow$ `fail`
  `(true,` $L_1, \ldots, L_k$) $\rightarrow (L_1, \ldots, L_k)$
  `(fail; G)` $\rightarrow$ `G`
  `var(X)` $\rightarrow$ `true` (falls `X` zum erstenmal vorkommt)
  Somit wird aus (3):
  (4) `([1|RL] = L3, append([], L2, RL))`
  Weitere partielle Auswertung des letzten Literals ergibt:
  (5) `([1|RL] = L3, L2 = RL)`
  Rückwärtsunifikation und Löschen: `[1|L2] = L3`
  Somit kann die Klausel

  ```
  p(L2, L3) ← append([1], L2, L3)
  ```

  zu dem Faktum

  ```
  p(L2, [1|L2])
  ```

  transformiert werden.

Allgemein benötigen partielle Auswerter eine gute Strategie, die die Terminierung und vernünftige Transformationen (Codegröße!) sicherstellen. Das Problem der partiellen Auswertung ist gerade das Finden solcher Strategien. Der Vorteil ist die semantische Äquivalenz vom ursprünglichen und transformierten Programm.

Wir werden die partielle Auswertung verwenden, um mit einer geschickten Strategie Prolog-Programme so zu transformieren, daß sie leicht in effiziente imperative Programme übersetzt werden können.

3.2 Explizite Unifikation

Mögliches Vorgehen: Werte das Programm eines Prolog-Interpreters zusammen mit einem Prolog-Programm partiell aus. Das Ergebnis ist ein effizientes, übersetztes Prolog-Programm. Dies wurde in [KC84] gemacht: Dort wurde ein in Lisp geschriebener Prolog-Interpreter partiell ausgewertet zu einem Lisp-Programm, das dem übersetzten Prolog-Programm entspricht. Die Effizienz der übersetzten Programme ist vergleichbar mit denen eines handgeschriebenen Compilers, nur die partielle Auswertung benötigt sehr viel Zeit.

Im folgenden werden wir die Methode von [Kur86] beschreiben, mit der man einige Instruktionen der „Warren Abstract Machine“ [War83] systematisch herleiten kann.

Grundsätzliches Vorgehen:

Gehe von einem implizit gegebenen Prolog-Interpreter aus (z.B. dem aus Kapitel 2). Dieser führt automatisch die notwendigen Unifikationen durch, verwaltet die Terme in speziellen Speicherbereichen und kontrolliert den Programmablauf.

1. Schritt: Mache die Unifikation explizit. Führe überall, wo Unifikationen notwendig sind, explizite Aufrufe einer Unifikationsprozedur ein. Die interne Unifikation des Interpreters wird nur noch als Zuweisung benutzt.

2. Schritt: Mache die Termdarstellung explizit. Definiere dazu einen Heap mit entsprechenden Operationen und führe überall, wo neue Terme erzeugt werden und auf alte Terme zugegriffen wird, explizite Operationen ein.

3. Schritt: Mache das Backtracking explizit. Das Ergebnis ist ein Prolog-Programm, das keine impliziten Mechanismen des Prolog-Interpreters ausnutzt und daher direkt in ein imperatives Programm übersetzt werden kann. Eine vorherige partielle Auswertung steigert die Effizienz wesentlich. Das Ergebnis der Übersetzung ist dann ein auf herkömmlichen Rechnerarchitekturen effizient ausführbares Programm.

Wir werden hier nur die Schritte 1 und 2 durchführen, um die prinzipielle Vorgehensweise aufzuzeigen und um den Prolog-Compiler im nächsten Kapitel zu motivieren.

Vermeidung der impliziten Unifikation durch Transformation jeder Programmklausel der Form

$$p(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_k$$

in die Form

$$p(X_1, \ldots, X_n) \leftarrow \texttt{unipp}(t_1, X_1), \ldots, \texttt{unipp}(t_n, X_n), L_1, \ldots, L_k$$

($X_1, \ldots, X_n$ sind paarweise verschiedene, neue Variablen). Dadurch wird beim Beweis von Literalen keine echte Unifikation, sondern nur noch eine Zuweisung an Variablen durchgeführt. Diese Variablen

werden später durch Maschinenregister realisiert (vgl. Kapitel 4). Jede Unifikation wird explizit durch einen `unipp`-Aufruf durchgeführt. Implementierung von `unipp` in Prolog:

```
unipp(S, T) ← atomic(S), (var(T), T := S;  T == S)
unipp(S, T) ← var(S), S := T
unipp(S, T) ← struct(S), functor(S, F, N),
                    (var(T), functor(T, F, N);
                     struct(T), functor(T, G, M), F==G, M==N),
                T =.. [_|TL], S =.. [_|SL],
                unipp_args(SL, TL)

unipp_args([], []) ←
unipp_args([S|SL], [T|TL]) ← unipp(S, T), unipp_args(SL, TL)
```

Anmerkungen:

- Diese Definition der Unifikation ist zugeschnitten auf unsere geplante Anwendung (partielle Auswertung).
- Der Einfachheit halber wird kein occur check durchgeführt.
- `:=/2` entspricht der normalen Unifikation, wobei das linke Argument immer eine ungebundene Variable ist. Somit wird diese Unifikation nur als Zuweisung verwendet.
- `atomic`, `var` bzw. `struct` sind erfüllt, wenn das Argument eine Konstante, eine Variable bzw. eine Struktur ist. Das Prädikat `functor` setzt eine Struktur mit dem führenden Funktor und dessen Stelligkeit in Relation, wohingegen `=..` zu einer Struktur die Liste der Argumente liefert (vgl. Standardprädikate von Prolog).
- `unipp` enthält ansonsten keine implizite Unifikation.

Anwendung auf `append`-Prädikat:

```
append([], L, L) ←
append([E|R], L, [E|RL]) ← append(R, L, RL)
```

`append`-Prädikat mit expliziter Unifikation:

```
append(X1, X2, X3) ← unipp([], X1), unipp(L, X2), unipp(L, X3)
append(X1, X2, X3) ← unipp([E|R], X1), unipp(L, X2), unipp([E|RL], X3),
                     append(R, L, RL)
```

Partielle Auswertung dieser Klauseln mit Klauseln für `unipp` ergibt (Auffalten der `unipp`-Literale, wenn erstes Argument bekannt ist):

```
append(X1, X2, X3) ← (var(X1), X1 := []; X1 == []), unipp(X2, X3)

append(X1, X2, X3) ← (var(X1), functor(X1, '.', 2);
```

```
 struct(X1), functor(X1, F, N), F=='.', N==2),
X1 =.. [_, X11, X12],
(var(X3), functor(X3, '.', 2);
 struct(X3), functor(X3, G, M), G=='.', M==2),
X3 =.. [_, X31, X32],
unipp(X11, X31),
append(X12, X2, X32)
```

Problem: Code wird viel größer, aber bestimmte Codesequenzen wiederholen sich mit verschiedenen Parametern.

Idee: Führe für diese Sequenzen spezielle Prädikate (Maschinenbefehle einer speziellen Prolog-Maschine) ein, die man leicht in einer niedrigen Programmiersprache implementieren kann:

```
unipp_const(C, T) ← var(T), T := C; T == C
unipp_struct(F, N, T, Args) ← (var(T), functor(T, F, N);
                                struct(T), functor(T, G, M), F==G, N==M),
                               T =.. [_|Args]
```

Neudefinition der Unifikation mit diesen speziellen Prädikaten:

```
unipp(S, T) ← atomic(S), unipp_const(S, T)
unipp(S, T) ← var(S), S := T
unipp(S, T) ← struct(S), functor(S, F, N),
              unipp_struct(F, N, T, TL),
              S =.. [_|SL], unipp_args(SL, TL)
```

Eine partielle Auswertung der `append`-Klauseln mit expliziter Unifikation mit den neuen Klauseln für `unipp` ergibt:

```
append(X1, X2, X3) ← unipp_const([], X1), unipp(X2, X3)
append(X1, X2, X3) ← unipp_struct('.', 2, X1, [X11, X12]),
                     unipp_struct('.', 2, X3, [X31, X32]),
                     unipp(X11, X31),
                     append(X12, X2, X32)
```

Es ist noch ein Aufruf des vollen Unifikationsalgorithmus' notwendig, weil zur Übersetzungszeit keine Information über `X11` und `X31` vorliegt. Aber in diesem Code ist keine implizite Unifikation mehr vorhanden.

3.3 Explizite Termrepräsentation

Die Speicherverwaltung für die Terme ist bisher noch implizit. Wir wollen dies jetzt explizit machen und geben dazu eine Implementierung der „structure copying"-Methode an.

Beim „structure copying" werden alle neu erzeugten Terme in einem speziellen Speicherbereich, dem sog. Heap, gespeichert. Jeder Speicherplatz im Heap besteht aus einem Etikett („tag") und einem

Wert. Je nach Art des gespeicherten Terms gibt es folgende Möglichkeiten:

Etikett	Wert
const	Konstante
free	Adresse dieser Speicherzelle
ref	Adresse der referenzierten Zelle
func	Funktorname/Stelligkeit

In der Praxis ist die Klassifizierung noch feiner (Unterscheidung bei Konstanten zwischen Atomen, ganzen Zahlen und Gleitkommazahlen).

Ein **Heap** besteht aus einer adressierbaren Folge von Speicherzellen des obigen Typs und einem **Heap-Zeiger**, der auf die nächste freie Speicherzelle im Heap verweist. In unserer Implementierung gibt es einen global bekannten Heap, der durch folgende Prädikate manipuliert werden kann:

heap`(Addr, Tag, Value)`: `Addr` muß mit der Adresse einer belegten Heapzelle instantiiert sein. Nach erfolgreichem Aufruf ist `Tag` an das Etikett und `Value` an den Wert der Speicherzelle mit Adresse `Addr` gebunden.

heap_entry`(Addr, Tag, Value)`: `Tag` und `Value` müssen instantiiert sein. Die nächste freie Speicherzelle wird mit Etikett `Tag` und Wert `Value` belegt, die Adresse dieser Speicherzelle wird mit `Addr` unifiziert und der Heap-Zeiger wird um eins erhöht.

heap_change`(Addr, Tag, Value)`: `Addr` muß mit der Adresse einer belegten Speicherzelle instantiiert sein. Diese Speicherzelle erhält das neue Etikett `Tag` und den neuen Wert `Value`.

Man beachte, daß wir an dieser Stelle noch nicht den Kontrollfluß (Backtracking) explizit betrachten, sondern die Prolog-Strategie als gegeben annehmen. Daher müssen auch die Prädikate zur Manipulation des Heap diese Strategie berücksichtigen, d.h. sie müssen **rücksetzbar** sein: Bei einem eventuellen Backtracking muß die Wirkung von `heap_entry` und `heap_change` (Veränderung des Heap) aufgehoben werden.

Zusammengesetzte Terme werden wie folgt gespeichert: Erst der Funktor, dann nacheinander alle Komponenten.

Beispiel:

`f([], X, X)` $\Longrightarrow$

1:	func	`f/3`
2:	const	`[]`
3:	free	3
4:	ref	3

`f(g(Z), Z)` $\Longrightarrow$

5:	func	`f/2`
6:	func	`g/1`
7:	free	7
8:	ref	7

Umwandlung eines Prolog-Terms in Heap-Darstellung:

- Darstellung von Heap-Adressen in Prolog-Programmen: $(\$h, A)$

- **Gemischte Terme** sind Terme, die wie Prolog-Terme aufgebaut sind und auch Heap-Adressen enthalten können, d.h. ($\$h, A$) ist **kein** Prolog-Term.
- Das Prädikat `heap_address(MT, A)` ist genau dann erfüllt, wenn `MT` eine Heap-Adresse der Form ($\$h$, `A`) ist.

Dann wandelt das Prädikat `rep(MT, Addr)` den (eventuell gemischten) Term `MT` in seine Heap-Repräsentation um, wobei diese auf dem Heap an der Adresse `Addr` beginnt.

```
rep(MT, Addr) ← atomic(MT), heap_entry(Addr, const, MT)
rep(MT, Addr) ← var(MT), heap_entry(Addr, free, Addr), MT := ($h,Addr)
rep(MT, Addr) ← heap_address(MT, A), heap_entry(Addr, ref, A)
rep(MT, Addr) ← struct(MT), functor(MT, F, N), heap_entry(Addr, func, F/N),
                MT =.. [_|Ts], rep_args(Ts)

rep_args([]) ←
rep_args([T|Ts]) ← rep(T,_), rep_args(Ts)
```

Jede Prolog-Variable wird also nur einmal auf dem Heap repräsentiert (2. Klausel). Kommt eine Prolog-Variable mehrfach in MT vor, dann werden die anderen Vorkommen durch Verweise (`ref`-Etikett) repräsentiert (3. Klausel).

Für die Unifikation mit Heap-Termen benötigen wir ein Prädikat `arg_addrs(Addr, Addrs)`, das uns die Liste `Addrs` der Adressen der Argumente des zusammengesetzten Terms, dessen Funktor bei `Addr` steht, liefert:

```
arg_addrs(Addr, Addrs) ← heap(Addr, func, F/N),
                         Addr1 is Addr+1,
                         addr_list(N, Addr1, Addrs)

addr_list(0, _, []) ←
addr_list(M, A, [A|As]) ← M > 0, next_addr(A, A1),
                         M1 is  M-1,
                         addr_list(M1, A1, As)

next_addr(A, A1) ← heap(A, Tag, Value),
                   next_addr_case(A, Tag, Value, A1)

next_addr_case(A, const, _, A1) ← A1 is A+1
next_addr_case(A, free, _, A1) ← A1 is A+1
next_addr_case(A, ref, _, A1) ← A1 is A+1
next_addr_case(A, func, F/N, A1) ← arg_addrs(A, As),
                                   last(As, LA),
                                   next_addr(LA, A1)

last(L, E) ← append(_, [E], L)
```

Dereferenzierung von Heap-Adressen (wird später benutzt):

```
deref(A, DA) ← heap(A, ref, Addr), !, deref(Addr, DA)
deref(A, A) ←
```

Beispiele: (vgl. oben)

```
arg_addrs(1, As) → As = [2, 3, 4]
arg_addrs(5, As) → As = [6, 8]
arg_addrs(6, As) → As = [7]
```

Idee des „structure copying“:

Bei Aufruf eines Literals stehen die Argumentterme auf dem Heap. Diese werden mit dem Klauselkopf (der nicht auf dem Heap steht) unifiziert. Wenn dabei Variablen an einen Teilterm des Klauselkopfes gebunden werden, wird dieser Teilterm auf den Heap kopiert.

Implementierung durch Prädikat `uniph(T, Addr)`, das einen gemischten Term (Argument des Klauselkopfes) mit einem Heap-Term nach der „structure copying“-Methode unifiziert (wenn bei Aufruf `Addr` instantiiert ist, wird `T` mit `Addr`-Term unifiziert, sonst wird `T` auf Heap bei Adresse `Addr` kopiert).

```
uniph(T, A) ← atomic(T), (var(A), heap_entry(A, const, T);
                          (deref(A, DA), heap(DA, Tag, Value),
                             (Tag == free, heap_change(DA, const, T);
                             (Tag == const, T == Value)))

uniph(T, A) ← var(T), (var(A), heap_entry(A, free, A); true)
              T := ($h,A)

uniph(T, A) ← heap_address(T, P), (var(A), heap_entry(A, ref, P);
                                   unihh(P, A))

uniph(T, A) ← struct(T), functor(T, F, N),
                          (var(A), heap_entry(A, func, F/N),
                                   get_var_list(N, As)
                          ;
                           deref(A, DA), heap(DA, Tag, Value),
                             (Tag == free, heap_entry(AF, func, F/N),
                                           heap_change(DA, ref, AF),
                                           get_var_list(N, As);
                              Tag == func, Value == F/N,
                                           arg_addrs(DA, As))
                          ),
                          T =.. [_|Ts],
                          uniph_args(Ts, As)
```

```
uniph_args([], []) ←
uniph_args([T|Ts], [A|As]) ← uniph(T, A), uniph_args(Ts, As)
```

Hierbei gilt:

- **get_var_list**`(N, VL)` unifiziert `VL` mit einer Liste der Länge `N`, deren Elemente paarweise verschiedene neue Variablen sind.
- **unihh**`(A1, A2)` unifiziert zwei Terme, die vollständig auf dem Heap liegen (Definition von `unihh` kann analog zu `uniph` erfolgen).
- Die Notwendigkeit der Dereferenzierungen (`deref`-Literale) werden wir später erläutern.

Wie schon erwähnt, sollen nun bei Aufruf eines Literals alle Argumente Heap-Terme sein, d.h. es werden bei Aufruf nur noch Heap-Adressen übergeben. Daher benötigen wir ein Prädikat `put(T, A)`, das einen Term auf den Heap schreibt, falls er dort noch nicht ist:

```
put(T, A) ← heap_address(T, A), !
put(T, A) ← rep(T, A)
```

Somit wird jedes Literal

$$p(t_1, \ldots, t_n)$$

im Rumpf einer Klausel durch das Ziel

$$\texttt{put}(t_1, B_1), \ldots, \texttt{put}(t_n, B_n), p(B_1, \ldots, B_n)$$

ersetzt. $B_1, \ldots, B_n$ sind paarweise verschiedene neue Variablen. Wenn wir noch berücksichtigen, daß die Unifikation weiterhin explizit bleibt, d.h. es werden `uniph`-Aufrufe für den Klauselkopf eingeführt, dann werden die `append`-Klauseln wie folgt transformiert:

```
append(A1, A2, A3) ← uniph([], A1), uniph(L, A2), uniph(L, A3)
append(A1, A2, A3) ← uniph([E|R], A1), uniph(L, A2), uniph([E|RL], A3),
                     put(R, B1), put(L, B2), put(RL, B3),
                     append(B1, B2, B3)
```

Partielle Auswertung mit `uniph`-Definition ergibt (dabei wird berücksichtigt, daß `append` nie mit Variablen aufgerufen wird):

```
append(A1, A2, A3) ← deref(A1, DA1), heap(DA1, Tag, Value),
                     (Tag == free, heap_change(DA1, const, []);
```

```
                          Tag == const, [] == Value),
                        unihh(A2, A3)

append(A1, A2, A3) ← deref(A1, DA1), heap(DA1, Tag1, Value1),
                        (Tag1 == free, rep([E|R], R1),
                              heap_change(DA1, ref, R1);
                         Tag1 == func, Value1 == '.'/2,
                           arg_addrs(DA1, As1), uniph_args([E,R], As1)),
                        deref(A3, DA3), heap(DA3, Tag3, Value3),
                        (Tag3 == free, rep([E|RL], R3),
                              heap_change(DA3, ref, R3);
                         Tag3 == func, Value3 == '.'/2,
                           arg_addrs(DA3, As3), uniph_args([E,RL]), As3)),
                        put(R, B1), put(($h,A2), B2), put(RL, B3),
                        append(B1, B2, B3)
```

Gleiche Idee wie bei expliziter Unifikation:

Definiere spezielle Instruktionen je nach Typ des zu unifizierenden Terms:

```
uniph_const(C, A) ← (var(A), heap_entry(A, const, C);
                        (deref(A, DA), heap(DA, Tag, Value),
                           (Tag == free, heap_change(DA, const, C);
                            Tag == const, C == Value)))

uniph_var(V, A) ← (var(A), heap_entry(A, free, A); true)

uniph_value(P, A) ← (var(A), heap_entry(A, ref, P); unihh(P, A))

uniph_struct(F/N, A, S) ← (var(A), heap_entry(A, func, F/N),
                                    get_var_list(N, S)
                               ;
                                deref(A, DA), heap(DA, Tag, Value),
                                    (Tag == free, heap_entry(AF, func, F/N),
                                                  heap_change(DA, ref, AF),
                                                  get_var_list(N, S);
                                     Tag == func, Value == F/N,
                                                  arg_addrs(DA, S)))
```

Damit kann `uniph` so definiert werden:

```
uniph(T, A) ← atomic(T), uniph_const(T, A)
uniph(T, A) ← var(T), uniph_var(T, A), T = ($h, A)
uniph(T, A) ← heap_address(T, P), uniph_value(P, A)
uniph(T, A) ← struct(T), functor(T, F, N),
              uniph_struct(F/N, A, S),
              T =.. [_|Ts],
```

```
                uniph_args(Ts, S)

uniph_args([], []) ←
uniph_args([T|Ts], [A|As]) ← uniph(T, A), uniph_args(Ts, As)
```

Partielle Auswertung der `append`-Klauseln mit dieser Neudefinition ergibt:

```
append(A1, A2, A3) ← uniph_const([], A1), uniph_value(A2, A3)
append(A1, A2, A3) ← uniph_struct('.'/2, A1, [A11, A12]),
                        uniph_var(E, A11),
                        uniph_var(R, A12),
                     uniph_struct('.'/2, A3, [A31, A32]),
                        uniph_value(E, A31),
                        uniph_var(RL, A32),
                     append(A12, A2, A32)
```

Anmerkungen:

- Bei dieser Version von `append` sind keine impliziten Unifikationen vorhanden und auch die Repräsentation von Termen ist explizit, da hier nur noch Heap-Terme, aber keine Prolog-Terme vorkommen (es kommen lediglich Namen von Prolog-Variablen in `uniph_var`-Literalen vor, auf die man bei der Implementierung verzichten kann).
- Die allgemeine Unifikation des Klauselkopfes wurde ersetzt durch spezielle Unifikationsinstruktionen, die die Form der zur Übersetzungszeit bekannten Terme berücksichtigen. Trotzdem ist noch eine komplette Unifikation auf Heap-Termen nötig, falls zur Übersetzungszeit nicht genug Information über die Terme vorhanden ist (siehe Definition von `uniph_value`).
- Da die allgemeine Unifikation durch spezialisierte Instruktionen ersetzt worden ist, sind nun zur Laufzeit weniger Abfragen notwendig, daher kann dieses (übersetzte Programm) effizienter ausgeführt werden.
- Die Klauseln könnten nun in eine imperative Programmiersprache übersetzt werden, falls man dort einen Backtracking-Mechanismus zur Verfügung hat. Andere Möglichkeit: Mache Backtracking explizit durch geschickte Zusatzprädikate und werte dies partiell aus. Ein Ansatz hierfür wird in [Nil90b] vorgeschlagen.
- Heutige Compiler übersetzen Prolog-Programme nach folgenden Prinzipien:
  - Generierung von speziellen Unifikationsinstruktionen, abhängig von den im Quellprogramm vorhandenen Termen.
  - Generierung von Instruktionen zur Kontrolle des Programmablaufs, d.h. zur Verwaltung des Backtrackings.

Das erste Prinzip haben wir in diesem Kapitel gezeigt und wir werden die hier angegebenen Instruktionen später wiederfinden. Das zweite Prinzip wird im nächsten Kapitel weiter vertieft. Neben dieser Übersetzung enthält jedes übersetzende Prolog-System ein Laufzeitsystem, das die Speicherstrukturen verwaltet, einen Unifikationsalgorithmus enthält etc.

Beispiel zur Notwendigkeit der Dereferenzierung:[1]
(die Dereferenzierung fehlt in [Kur86])

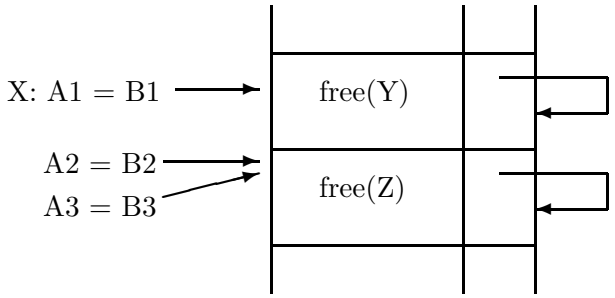
```
p(X, X, a) ←
q ← p(Y, Z, Z)
```

Klauseln mit expliziter Unifikation und Termrepräsentation:

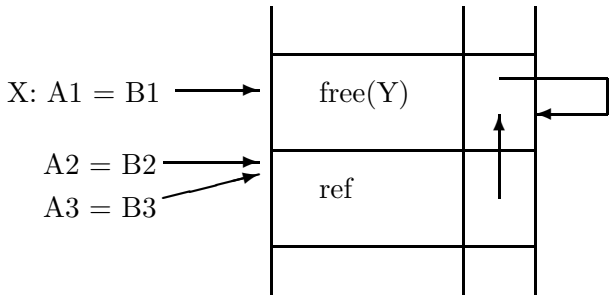
```
p(A1, A2, A3) ← uniph(X, A1), uniph(X, A2), uniph(a, A3)
q ← put(Y, B1), put(Z, B2), put(Z, B3), p(B1, B2, B3)
```

Zu beweisen ist das Ziel: `q`

Heap-Situation vor Aufruf des Literals `uniph(X,A2)`:



Zum Beweis von `uniph(X,A2)` muß `unihh(A1,A2)` bewiesen werden. Wir nehmen an, daß hierdurch die freie Speicherzelle `A2` an `A1` gebunden wird (für den umgekehrten Fall muß man das Beispiel leicht modifizieren). Dann ergibt sich folgende Heap-Situation vor Aufruf von `uniph(a,A3)`:



Nun verweist beim folgenden `uniph(a, A3)`-Aufruf der Heap-Term-Zeiger auf eine `ref`-Heapzelle, d.h. im allgemeinen muß immer zuerst dereferenziert werden.

[1] Dieses Beispiel hat Jörg Süggel gefunden.

In diesem Kapitel sollte klar geworden sein, was ein Compiler im Vergleich zum Interpreter prinzipiell machen kann, um den Ablauf des Programms zu beschleunigen. Dies soll die Motivation für das nächste Kapitel sein, in der eine heute weit verbreitete Übersetzungstechnik dargestellt wird.

Kapitel 4

Die „Warren Abstract Machine“ (WAM)

Wichtige Literatur: [War83] [GLLO85] [War77] [TW84] [Pro87] [AK91] [Han88]

In diesem Kapitel wird die WAM [War83] vorgestellt, eine virtuelle Maschine zur effizienten Ausführung von Prolog-Programmen, die eine Weiterentwicklung der in [War77] vorgestellten Maschine ist. Sie weist außerdem viele Ähnlichkeiten zu der in [Clo85a] vorgestellten Implementierung auf.

Die WAM stellt eine Menge einfacher Instruktionen (WAM-Maschineninstruktionen) zur Verfügung, die überwiegend sehr effizient auf herkömmlichen Rechnerarchitekturen ausgeführt werden können. Jedes Prolog-Programm muß somit in ein WAM-Maschinenprogramm übersetzt werden. Ein **WAM-Maschinenprogramm** kann entweder

- emuliert (dazu werden die WAM-Instruktionen als Bytecodes dargestellt und von einem in C, Assembler oder Mikrocode implementierten Emulator interpretiert),
- in Maschinencode übersetzt, oder
- auf einer speziellen Hardware, die WAM-Instruktionen als Maschinencode hat, ausgeführt werden.

Neben dem WAM-Maschinenprogramm ist zur Ausführung noch ein **Laufzeitsystem** erforderlich, das die komplexen Operationen realisiert (Speicherverwaltung, Unifikation) und die Implementierung der vordefinierten Prädikate enthält.

Eine solche Realisierung ist die Grundlage der meisten heutigen Prolog-Implementierungen (abgesehen von Interpretern), wobei der WAM-Code entweder emuliert oder in Maschinencode übersetzt wird. Daher werden wir darauf jetzt weiter im Detail eingehen.

Grundprinzipien der WAM:

- Zur Verwaltung während der Laufzeit erzeugter Terme wird die „structure copying“-Methode verwendet.

- Für Klauseln wird die „structure sharing“-Methode verwendet, wobei die Bindungen der Variablen in einer Klausel in einer Umgebung und die Klausel selbst als Sequenz von WAM-Instruktionen repräsentiert werden.
- Bei den auf dem (lokalen) Stack gespeicherten Datenstrukturen wird unterschieden zwischen Umgebungen und Backtrackpunkten. Durch diese Trennung ist es möglich, sowohl Umgebungen als auch Backtrackpunkte erst bei Bedarf anzulegen.
- Bei deterministischen Berechnungen wird die Umgebung immer so verkürzt, daß nur noch der real beanspruchte Platz reserviert wird. Dies entspricht einer Verallgemeinerung von „tail recursion optimization“.
- Viele Entscheidungen, die Einfluß auf die Größe des zur Laufzeit benötigten Speicherplatzes haben, werden erst zur Laufzeit gefällt (Anlegen eines Backtrackpunkts, Speichern von Variablen auf dem globalen Stack). Hierdurch wird, im Gegensatz zum Fällen der Entscheidung zur Übersetzungszeit, in der Regel sehr viel Speicherplatz eingespart.
- Um schnell auf passende Klauselköpfe zuzugreifen und um Backtrackpunkte nur bei Bedarf anzulegen (falls ein Prädikat durch viele Klauseln definiert ist), wird für die Folge aller Klauseln zu einem Prädikat eine bestimmte Indexstruktur angelegt.
- Die Unifikation von Literalen mit Klauselköpfen ist entsprechend dem in Kapitel 3 vorgestellten Schema auscompiliert, d.h. es gibt spezielle Unifikationsanweisungen für spezielle Datentypen. Das Schema ist gegenüber Kapitel 3 noch verfeinert. Zusätzlich gibt es Anweisungen, die den Kontrollfluß zur Laufzeit regeln.

4.1 Speicherbereiche und Register

Alle zur Laufzeit relevanten Informationen werden in exakt definierten Speicherbereichen oder in speziellen Registern verwaltet. Diese Speicherbereiche und Register enthalten in der Regel Prolog-Terme. Ein **Prolog-Term** wird dargestellt als ein Paar bestehend aus einem Etikett („tag“) und einem Wert. Das Etikett gibt den Typ des Terms an, der Wert enthält je nach Etikett eine unterschiedliche Information. Hierbei wird mindestens zwischen folgenden Typen unterschieden:

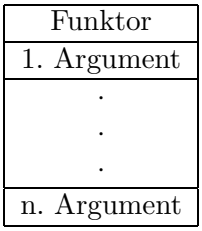
Etikett	**Wert**
reference	Adresse
structure	Adresse der Struktur
list	Adresse der (nicht-leeren) Liste
integer	Wert der ganzen Zahl
atom	Nummer des Atoms

Die **Atome** (und **Funktoren**) sind durchnumeriert. Alternativ könnte man (im Hinblick auf das vordefinierte Prädikat `name/2`) auch einen Speicherbereich reservieren, in dem die textuellen Darstellungen der Atome gespeichert sind. In diesem Fall enthielte der Wert eines atom-Terms die Anfangsadresse

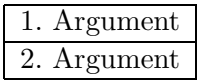
der textuellen Darstellung des Atoms. Wichtig für die Unifikation ist lediglich, daß die Gleichheit von Atomen direkt durch die Gleichheit des Wertfelds entschieden werden kann.

Bei **Variablen** wird zunächst nicht unterschieden zwischen ungebundenen und gebundenen Variablen, sondern bei ungebundenen Variablen enthält das Wertfeld einen Verweis auf sich selbst.

Bei **Strukturen** werden die **Listen** durch ein spezielles Etikett unterschieden, da diese sehr häufig vorkommen. Dies hat den Vorteil, daß der Funktor `'.'/2` bei Listen nicht explizit gespeichert werden muß. Somit verweist das Wertfeld eines structure-Terms auf das erste Element der folgenden Darstellung:



Bei einem list-Term wird dagegen auf folgende Darstellung verwiesen:



Die Argumente sind bei beiden Darstellung wieder Prolog-Terme, bestehend aus Etikett und Wert.

Mögliche Implementierung der Darstellung eines Prolog-Terms: 1 Maschinenwort, wobei das Etikett in den ersten oder letzten Bits codiert ist.

Die WAM enthält folgende **Speicherbereiche**:

Codebereich: Dieser Bereich enthält die WAM-Instruktionen des auszuführenden Programms.

(Lokaler) Stack: Dieser Stack enthält die zwei folgenden Arten von Objekten:

Umgebungen: Ein Feld für Variablen, die in der Klausel vorkommen (jedes Element enthält die Darstellung eines Prolog-Term, s.o.), zusammen mit einem Verweis in den Rumpf einer anderen Klausel (Ziele, die noch bewiesen werden müssen) und einem Verweis auf eine andere Umgebung. Eine Umgebung wird angelegt, wenn eine Klausel abgearbeitet wird, in deren Rumpf mehr als ein Literal vorkommt.

Backtrackpunkte: Diese enthalten alle Informationen, die notwendig sind, um einen früheren Berechnungszustand (im Falle von Backtracking) wiederherzustellen. Ein Backtrackpunkt wird angelegt, wenn ein Literal mit einer Klausel bewiesen wird und es noch andere, eventuell passende Klauseln gibt.

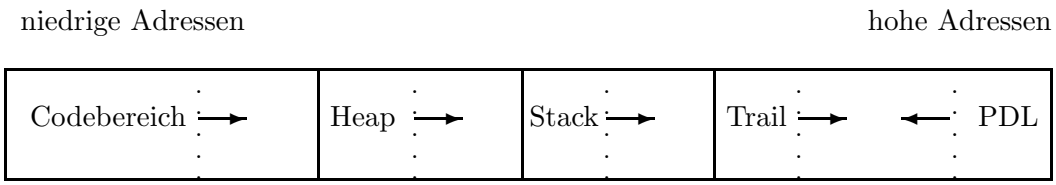
Der Stack wird verkleinert beim Backtracking, bevor das letzte Ziel in einer deterministischen Klausel aufgerufen wird und bei einem Cut (s.u.).

Heap: Dieser Stack (manchmal auch **globaler Stack** genannt) enthält neu erzeugte Terme, die beim „structure copying“ entstehen (also Strukturen und Listen), und evtl. ungebundene Variablen. Er schrumpft nur beim Backtracking (und bei einer evtl. implementierten Speicherbereinigung auf dem Heap).

Trail: Dieser Stack enthält alle Variablen, die während einer Unifikation gebunden worden sind und die beim Backtracking wieder auf „ungebunden“ gesetzt werden müssen. Er schrumpft nur beim Backtracking.

PDL: Dieser kleine Stack wird für die rekursive Implementierung der Unifikation benutzt. Bei Implementierungen auf konventionellen Architekturen (z.B. 680x0) kann seine Funktion durch den Systemstack übernommen werden.

Diese Speicherbereiche können z.B. so angeordnet werden:



Wir gehen davon aus, daß alle Stacks (bis auf PDL) von den niedrigen zu den hohen Adressen wachsen, denn dann kann man eine einfache Strategie angeben, die verhindert, daß Zeiger bei Speicheroptimierungen in undefinierte Datenbereiche zeigen: Wenn *eine Variable an eine andere gebunden* wird, dann muß der Zeiger von der hohen zu der niedrigen Adresse zeigen. Insbesondere darf kein Zeiger vom Heap in den Stack zeigen, da der Stack schon nach Abarbeitung von deterministischen Klauseln verkleinert wird, während Heap-Elemente erst beim Backtracking gelöscht werden.

Der aktuelle Berechnungszustand der WAM ist definiert durch den Inhalt der o.g. Speicherbereiche und der folgenden **Register**, die Verweise in diese Speicherbereiche oder Terme enthalten:

Bezeichner	Name	Bedeutung
`P`	program pointer	Verweis auf die nächste auszuführende WAM-Instruktion im Codebereich
`CP`	continuation program pointer	Verweis auf das nächste Ziel im Codebereich, das bewiesen werden muß, wenn der Rumpf der aktuellen Klausel (in die P zeigt) bewiesen worden ist
`E`	last environment	Verweis auf die zuletzt angelegte Umgebung (oder aktuelle Umgebung) im lokalen Stack
`B`	last backtrack point	Verweis auf den zuletzt angelegten Backtrackpunkt im lokalen Stack
`TR`	top of trail	Nächstes freies Trail-Element
`H`	top of heap	Nächstes freies Heap-Element
`S`	structure pointer	Verweis auf die Argumente einer Struktur im Heap, die unifiziert werden müssen
`RW`	read/write register	Zeigt an, ob eine Unifikation von Strukturen im Lese- oder Schreibmodus erfolgt. Im Schreibmodus wird die Struktur auf dem Heap gespeichert („structure copying“)
`A1, A2, ...`	argument registers	Enthalten die Argumente des aktuell bearbeiteten Literals
`X1, X2, ...`	temporary variables	Variablen in der aktuellen Klausel, die nicht in der Umgebung gespeichert werden müssen

Somit besteht eine **Umgebung** aus folgenden Komponenten:

`EE`:	Letzter Wert von `E`
`ECP`:	Letzter Wert von `CP`
`Y1, Y2, ...`:	Permanente Variablen, d.h. Variablen in der aktuellen Klausel, die nicht temporär sind (s.u.).

Ein **Backtrackpunkt** besteht aus folgenden Komponenten:

`BA1, ...`, `BAn`:	Gesicherte Werte der Argumentregister
`BCP`:	Gesicherter Wert von `CP`
`BCE`:	Gesicherter Wert von `E`
`BB`:	Gesicherter Wert von `B`
`BTR`:	Gesicherter Wert von `TR`
`BH`:	Gesicherter Wert von `H`
`BP`:	Adresse der nächsten alternativen Klausel

Bei den Variablen, die in einer Klausel vorkommen, wird zwischen *temporären* und *permanenten* Variablen unterschieden, um Speicherplatz auf dem lokalen Stack zu sparen, denn nur die permanenten Variablen werden in einer Umgebung gespeichert. Warren gibt folgende Klassifikation an:

Eine Variable in einer Klausel ist **temporär**, wenn gilt:

1. Die Variable tritt zum ersten Mal im Kopf der Klausel, in einer Struktur oder im letzten Literal des Rumpfes auf.

2. Die Variable kommt nicht in zwei verschiedenen Literalen des Rumpfes vor.

3. Wenn die Variable im Klauselkopf vorkommt, dann kommt sie höchstens noch im ersten Literal des Klauselrumpfes vor.

Eine Variable in einer Klausel ist **permanent**, wenn sie nicht temporär ist. Permanente Variablen müssen in einer Umgebung auf dem lokalen Stack gespeichert werden. Sie werden mit `Y1`, `Y2`, ... bezeichnet. Sie werden nach ihrem letzten Auftreten im Rumpf geordnet, d.h. wenn `Yi` und `Yj` permanente Variablen mit `i<j` sind, dann kommt `Yi` hinter dem letzten Auftreten von `Yj` im Rumpf vor. Diese Sortierung wird dazu benutzt, um den Speicherplatz für die Umgebungen so klein wie möglich zu halten. Dies wird erreicht, indem der Speicherplatz einer Variablen in einer Umgebung freigegeben wird, sobald das Literal aufgerufen wird, in dem sie zum letzten Mal vorkommt. Wenn die Variable aber zu diesem Zeitpunkt noch ungebunden ist, muß für diese Variable eine Speicherzelle im Heap angelegt werden, falls dies noch nicht der Fall ist. Solche Variablen heißen unsicher. Eine Variable ist somit **unsicher** genau dann, wenn sie permanent ist und ihr erstes Auftreten nicht im Klauselkopf oder in einer Struktur liegt (im letzten Fall würde sie auf jeden Fall schon im Heap gespeichert sein, im ersten Fall im Heap oder eventuell in einer älteren Umgebung).

Beispiel: `p(X)` ← `q(X, f(Y), Z), r(Y), s(Z)`

`X` ist temporär, `Y` ist permanent und `Z` ist permanent und unsicher. Wenn diese Klausel zum Beweis eines `p`-Literals benutzt wird, dann wird eine Umgebung erzeugt, die die Variablen `Y` und `Z` enthält. Dabei ist `Y` die zweite und `Z` die erste permanente Variable, da `Z` noch nach `Y` im Rumpf vorkommt. Vor Aufruf von `r(Y)` kann der Speicherplatz für `Y` in der Umgebung freigegeben werden, da `Y` danach nicht mehr vorkommt. Vor Aufruf von `s(Z)` kann die gesamte Umgebung gelöscht werden. Falls `Z` jedoch beim Aufruf von `s(Z)` noch ungebunden ist, existiert `Z` nur in der aktuellen Umgebung. Da dieser Speicherplatz aber freigegeben werden soll (Prinzip der „tail recursion optimization“), muß für `Z` ein Speicherplatz im Heap angelegt werden und bei Aufruf von `s(Z)` enthält das Argument einen Verweis auf diesen Speicherplatz.

4.2 Instruktionssatz

Im folgenden beschränken wir uns auf die Beschreibung der Übersetzung aller zu *einem* Prädikat gehörigen Klauseln. Ein Prolog-Programm wird dann übersetzt, indem nacheinander alle in ihm definierten Prädikate mit ihren Klauseln übersetzt werden.

Bei der Übersetzung der Klauseln für ein Prädikat werden zwei Arten von Instruktionen generiert:

- Instruktionen, die der Übersetzung einer Klausel (Klauselkopf, Klauselrumpf) entsprechen.
- Instruktionen, die die einzelnen Klauseln in ihrer vorgegebenen Reihenfolge miteinander verbinden (Indizierungsinstruktionen).

4.2.1 Instruktionen zur Übersetzung einer Klausel

Wir besprechen zunächst die Übersetzung einer einzelnen Klausel. Grundsätzlich wird in der WAM folgendes Prinzip eingehalten: Bei Aufruf (Beweis) eines Literals mit n Argumenten sind die aktuellen Argumente immer in den Registern `A1,...,An` gespeichert. In diesen Registern befinden sich die oben angegebenen Repräsentationen der Prolog-Terme, d.h. Konstanten befinden sich direkt in den Registern, ansonsten ist dort eine Referenz auf einen Term im Heap oder auf ein Element in einer Umgebung gespeichert. Somit sind die grundsätzlichen Übersetzungstechniken schon in Kapitel 3 eingeführt worden.

- Wenn der Klauselkopf die Form $p(t_1, \ldots, t_n)$ hat, so wird er in folgende Instruktionen übersetzt:

  $$get(t_1, \texttt{A1}), \ldots, get(t_n, \texttt{An}).$$

  Die **get-Instruktionen** entsprechen den `uniph`-Literalen, d.h. $get(t_i, \texttt{Ai})$ unifiziert den aktuellen Term in Register `Ai` mit dem Prolog-Term t_i. Weil der Term t_i zur Übersetzungszeit vollständig bekannt ist, kann diese Instruktion spezialisiert werden zu Unifikationsinstruktionen mit speziellen Aufgaben (vgl. Abschnitt 3.3). In der WAM gibt es daher keine allgemeine Instruktion `get`, sondern nur spezielle Instruktionen wie `get_variable`, `get_constant` etc.

- Ein Literal im Klauselrumpf der Form $q(t_1, \ldots, t_n)$ wird in folgende Instruktionen übersetzt:

  $$put(t_1, \texttt{A1}), \ldots, put(t_n, \texttt{An}).$$

  Die Instruktion $put(t_i, \texttt{Ai})$ schreibt die Repräsentation des Prolog-Terms t_i in das Register `Ai` und evtl. auf den Heap (da in der WAM ja „structure copying“ benutzt wird). Weil der Term t_i schon zur Übersetzungszeit bekannt ist, gibt es in der WAM wie bei den `get`-Instruktionen keine allgemeine `put`-Instruktion, sondern nur spezielle.

Eine Klausel wird nun übersetzt, indem der Kopf und die Rumpfliterale in `get`- bzw. `put`-Instruktionen übersetzt werden, und dazwischen werden **Kontrollinstruktionen** eingefügt. Dies geschieht nach folgendem Schema, wobei nach Art der Klausel unterschieden wird:

Art:	**Faktum**	**Rumpf (ein Literal)**	**Rumpf (mehrere Literale)**
Klausel:	$P \leftarrow$	$P \leftarrow Q$	$P \leftarrow Q, R, S$

Übersetzung:	*get-Instr. für P*
	`proceed`
		get-Instr. für P
		put-Instr. für Q
		`execute Q`
			`allocate`
			get-Instr. für P
			put-Instr. für Q
			`call Q, N`
			put-Instr. für R
			`call R, N1`
			put-Instr. für S
			`deallocate`
			`execute S`

Die Bedeutung der einzelnen Kontrollinstruktionen ist (für alle nachfolgenden WAM-Instruktionen gilt immer, daß jede Instruktion nach ihrer Ausführung den Programmzeiger `P` um eine Instruktion erhöht, falls `P` in der Instruktion nicht explizit gesetzt wird):

proceed Diese Instruktion steht am Ende eines Faktums. An diesem Punkt muß das nächste Ziel (der Vaterklausel) bewiesen werden. Aktion:

```
P := CP
```

execute `Pr` Diese Instruktion steht am Ende des letzten Literals in einer Klausel. Der Programmzeiger wird auf den Code des Prädikats `Pr` gesetzt.

```
P := Pr
```

(wir gehen davon aus, daß der Beginn des WAM-Codes für jedes Prädikat durch eine Marke mit dem Namen des Prädikats gekennzeichnet ist).

allocate Diese Instruktion steht zu Beginn der Klausel, deren Rumpf mehr als ein Literal enthält. Sie generiert auf dem lokalen Stack Speicherplatz für die Umgebung und sichert die aktuellen Werte von `CP` und `E` in der neuen Umgebung.

```
Ce    := E
E     := top_of_stack
E.ECP := CP
E.EE  := Ce
```

(`top_of_stack` kann aus den Werten von `E`, `B` und dem letzten Aufruf berechnet werden, da an der Adresse `CP-1` immer der Befehl `call Pr,N` steht, wobei `N` die Größe der aktuellen Umgebung ist, s.u.).

call `Pr, N` Diese Instruktion ruft ein Literal im Klauselrumpf auf, nachdem die Argumentregister geladen worden sind, d.h. der Fortsetzungszeiger `CP` wird auf den Befehl hinter dieser Instruktion gesetzt und der Programmzeiger `P` auf den Code des aufgerufenen Prädikats. `N` gibt dabei die Anzahl der permanenten Variablen an, die nach diesem Literal in der aktuellen Umgebung noch benötigt werden, d.h. im obigen Schema ist `N1` $\leq$ `N`. Diese Angabe wäre prinzipiell zwar nicht notwendig, aber dadurch ist es möglich, schon während der Abarbeitung der Klausel die Umgebung auf dem lokalen Stack zu verkleinern (siehe später). Aktion:

```
CP := < Codeadresse nach dieser Anweisung >
P  := Pr
```

deallocate Diese Instruktion wird unmittelbar vor dem Aufruf des letzten Literals in einer Klausel mit mehreren Rumpfliteralen ausgeführt. Sie löscht die aktuelle Umgebung, was unter bestimmten Bedingungen der „tail recursion optimization“ entspricht (siehe später):

```
CP := E.ECP
```

```
E  := E.EE
```

Damit ist das Schema für die Übersetzung einer Klausel beschrieben. Wir wollen jetzt genauer auf die `get`- und `put`-Instruktionen eingehen. Zur Generierung dieser Instruktionen müssen zunächst die in der Klausel vorkommenden Variablen als temporär oder permanent (und unsicher) klassifiziert und durchnumeriert werden (vgl. letzten Abschnitt insbesondere zur Numerierung der Variablen). Bei der folgenden Beschreibung werden diese Bezeichnungen benutzt:

Bezeichnung	**Bedeutung**
`Ai`	Argumentregister
`Xi`	Temporäre Variablen
`Yi`	Permanente Variablen
`Vi`	Temporäre oder permanente Variable (`Xi` oder `Yi`)
`C`	Konstanten
`F`	Funktoren

Die folgenden Instruktionen sind nun die oben erwähnten spezialisierten `get`-Instruktionen:

Instruktion	**Bedeutung**
`get_variable Vn, Ai`	Unifiziere i. Argument mit der ungebundenen Variablen `Vn`. Diese Instruktion wird benutzt, wenn das i. Argument im Kopf eine Variable ist, die zum ersten Mal im übersetzten Code vorkommt.
`get_value Vn, Ai`	Unifiziere i. Argument mit der gebundenen Variablen `Vn`. Diese Instruktion wird benutzt, wenn das i. Argument im Kopf eine Variable ist, die schon einmal vorgekommen ist.
`get_constant C, Ai`	Unifiziere i. Argument mit der Konstanten `C`.
`get_nil Ai`	Unifiziere i. Argument mit der leeren Liste.
`get_structure F, Ai`	Unifiziere i. Argument mit einer Struktur mit Funktor `F`.
`get_list Ai`	Unifiziere i. Argument mit einer nicht-leeren Liste.

Für alle `get`-Instruktionen gilt: Wenn die Unifikation fehlschlägt, dann wird implizit die Instruktion „`fail`“ (s.u.) ausgeführt. Wenn während der Unifikation Variablen gebunden werden, dann wird für diese implizit die Instruktion „`trail`“ (s.u.) ausgeführt.

Beispiel zur Übersetzung eines Klauselkopfes:

Klausel: `p(X,Y,a,X)` ←

WAM-Code:

```
get_variable X1, A1
get_variable X2, A2
get_constant a, A3
get_value X1, A4
proceed
```

Die Instruktion `get_structure` (bzw. `get_list`) unifiziert ein Argument mit dem Hauptfunktor einer Struktur. Direkt danach müssen Instruktionen zur Unifikation der Argumente der Struktur folgen

(sog. `unify`-Instruktionen). Nach der `get_structure`-Anweisung arbeitet die WAM in zwei möglichen Zuständen weiter (Register `RW`): Im **Lese-Modus** ist das aktuelle Argument eine (auf dem Heap) existierende Struktur. Das Register `S` wird auf die 1. Komponente dieser Struktur gesetzt, und jede der nachfolgenden unify-Instruktionen bezieht sich auf den durch `S` adressierten Term und erhöht `S` um 1. Im **Schreib-Modus** ist das aktuelle Argument eine ungebundene Variable. Da diese infolge der Unifikation an die Struktur gebunden werden muß, wird die Struktur auf den Heap kopiert („structure copying"). Dazu wird zunächst an der Adresse `H` auf dem Heap eine Struktur mit dem Funktor `F` angelegt, die ungebundene Variable wird an diese Speicherzelle gebunden, und `H` wird um 1 erhöht. Jede der nachfolgenden `unify`-Instruktionen schreibt an der Stelle `H` ein Argument und erhöht `H` um 1.

Die **unify-Instruktionen** entsprechen den Argumenten einer Struktur. Ihnen geht immer eine `get_structure`, `get_list`- (oder `put_structure`, `put_list`-, s.u.) Instruktion voraus, die den Modus bestimmt. Je nach Modus wird ein Argument der Instruktion durch `S` bzw. `H` adressiert:

Instruktion	**Bedeutung**
`unify_void N`	Unifiziere die nächsten `N` Argumente mit nur einmal vorkommenden Variablen, d.h. erhöhe `S` um `N` (im Lese-Modus) bzw. schreibe `N` ungebundene Variablen auf den Heap (im Schreib-Modus). Diese Anweisung dient zur effizienten Implementierung von Strukturen der Form `f(_, _, _, X)` o.ä.
`unify_variable Vn`	Unifikation mit der ungebundenen Variablen `Vn`
`unify_value Vn`	Unifikation mit der gebundenen Variablen `Vn`
`unify_local_value Vn`	Unifikation mit der gebundenen Variablen `Vn`, die aber nicht unbedingt global (d.h. auf dem Heap) sein muß (global wäre sie, wenn sie z.B. mit einer `unify_variable`-Instruktion initialisiert worden wäre)
`unify_constant C`	Unifikation mit der Konstanten `C`
`unify_nil`	Unifikation mit der leeren Liste

Da in der Darstellung von Strukturen auf dem Heap die Argumente direkt hintereinander folgen, müssen **Unterstrukturen** mit Verweisen implementiert werden: Wenn eine Unterstruktur `f(...)` in einer anderen Struktur *St* vorkommt, dann wird in der Folge der `unify`-Instruktionen der Struktur *St* für die Unterstruktur `f(...)` die Anweisung `unify_variable Xn` generiert (wobei `Xn` eine neue temporäre Variable ist), und hinter den `unify`-Anweisungen für *St* folgen dann die Anweisungen für die Unterstruktur `f(...)`, eingeleitet durch die Anweisung `get_structure f, Xn` (analog für Listenstrukturen). D.h. wir lassen bei `get_structure` auch temporäre Variablen zu, was aber keine konzeptuelle Erweiterung ist, da man die `A`- und `X`-Register identifizieren kann (vgl. Kapitel 4.4.1).

Beispiele zur Übersetzung von Strukturen in Klauselköpfen (bei Funktoren wird auch immer die Stelligkeit mit angegeben):

Klausel: `p(X, f(Y), g(X))` ←

WAM-Code:

```
get_variable X1, A1
get_structure f/1, A2
unify_void 1
```

```
get_structure g/1, A3
unify_local_value X1
proceed
```

In diesem Beispiel ist die spezielle Anweisung `unify_local_value` notwendig: Falls das dritte Argument im Schreib-Modus unifiziert wird, muß der Term `g(X)` auf den Heap geschrieben werden. Falls aber `X` eine Variable im lokalen Stack ist, kann der Wert von `X` (eine Referenz) nicht auf den Heap geschrieben werden, weil dann ein Verweis vom Heap in den Stack entstehen würde. Daher muß zuerst auf dem Heap eine ungebundene Variable als Argument von `g` geschrieben werden, und anschließend wird die Variable `X` im lokalen Stack an diese neue Variable im Heap gebunden. Genau dies macht `unify_local_value`, während dagegen `unify_value` den Wert von `X` direkt auf den Heap schreibt.

Das nächste Beispiel zeigt die Übersetzung von geschachtelten Strukturen:

Klausel: `p(f(g(X), X))` ←

WAM-Code:

```
get_structure f/2, A1
unify_variable X2
unify_variable X1
get_structure g/1, X2
unify_value X1
proceed
```

Die `put`-Instruktionen werden bei der Übersetzung von Literalen im Klauselrumpf verwendet. Sie laden Prolog-Terme in die Argumentregister `Ai`, im Gegensatz zu den `get`-Instruktionen, die die in den Argumentregistern stehenden Terme mit anderen unifizieren. Die `put`-Instruktionen sind analog zu den `get`-Instruktionen spezialisiert auf die verschiedenen möglichen Terme (es gibt jedoch noch eine zusätzliche Instruktion `put_unsafe_value` zum Laden unsicherer Variablen):

Instruktion	**Bedeutung**
`put_variable Vn, Ai`	Lade ungebundene Variable `Vn` in das Register `Ai`. Da diese Instruktion benutzt wird, wenn `Vn` hier zum ersten Mal auftritt, muß zunächst ein Speicherplatz mit der ungebundenen Variablen initialisiert werden: Wenn `Vn` permanent ist, d.h. `Vn = Yn`, dann wird `Yn` als ungebunden initialisiert und in `Ai` wird eine Referenz auf `Yn` geladen. Wenn `Vn` dagegen temporär ist, d.h. `Vn = Xn`, dann wird auf dem Heap eine neue, ungebundene Variable erzeugt und in `Xn` und `Ai` wird eine Referenz zu dieser Variablen gespeichert.
`put_value Vn, Ai`	Lade gebundene Variable `Vn` als i. Argument. Diese Instruktion wird benutzt, wenn das i. Argument des Rumpfliterals eine Variable ist, die hier nicht zum ersten Mal vorkommt, und wenn `put_unsafe_value` nicht benutzt wird (s.u.).
`put_unsafe_value Yn, Ai`	Lade permanente Variable in Register `Ai`. Diese Instruktion wird anstelle von `put_value` beim ersten Auftreten einer unsicheren permanenten Variablen als i. Argument *im letzten Literal* des Klauselrumpfes benutzt. Da das erste Auftreten einer unsicheren Variablen nicht im Klauselkopf oder in einer Struktur liegt, wurde sie durch eine `put_variable`-Anweisung initialisiert, d.h. es handelt sich evtl. um eine noch ungebundene Variable in der aktuellen Umgebung. Da der Speicherplatz von `Yn` vor dem Aufruf des Literals evtl. freigegeben werden soll ("tail recursion optimization", vgl. Abschnitt 4.1, Ende), muß diese Instruktion folgendes bewirken: Dereferenziere zunächst `Yn` und lade das Ergebnis in Register `Ai`. Falls das Ergebnis eine Variable in der aktuellen Umgebung ist, wird diese Variable an eine neu erzeugte, ungebundene Variable auf dem Heap gebunden und `Ai` wird ebenfalls an die neue Heap-Variable gebunden (beachte die Analogie zu der `unify_local_value`-Instruktion).
`put_constant C, Ai`	Lade Konstante `C` als i. Argument.
`put_nil Ai`	Lade die leere Liste als i. Argument.
`put_structure F, Ai`	Lade die Struktur mit Funktor `F` als i. Argument, d.h. erzeuge neue Struktur mit Funktor `F` auf dem Heap, lade in `Ai` eine Referenz auf die neue Struktur und arbeite im Schreib-Modus weiter.
`put_list Ai`	Lade eine nicht-leere Liste als i. Argument.

Für `put_structure` (bzw. `put_list`) gelten die Anmerkungen bei den `get`-Instruktionen analog: Hinter einer `put_structure`-Instruktion folgen `unify`-Instruktionen für die Argumente einer Struktur, die hier natürlich *immer im Schreib-Modus* ausgeführt werden. *Unterstrukturen* werden wie bei `get` durch eine Anweisung `unify_value Xn` übersetzt, wobei dann die Unterstrukturen mit `put_structure F, Xn` *vor* der Hauptstruktur aufgebaut werden.

Damit haben wir jetzt alle WAM-Instruktionen zur Übersetzung von einzelnen Klauseln kennengelernt. Bevor wir zu den Indizierungsinstruktionen zur Übersetzung von Klauselköpfen kommen, wollen wir noch einige Beispiele betrachten.

Beispiele:

Klausel: `append([], L, L)` ←

Übersetzung:

```
get_nil A1
get_variable X1, A2
get_value X1, A3
proceed
```

Klausel: `append([E|R], L, [E|RL])` ← `append(R, L, RL)`

Übersetzung:

```
get_list A1
unify_variable X1     % X1 = E
unify_variable X2     % X2 = R
get_variable X3, A2   % X3 = L
get_list A3
unify_value X1
unify_variable X4     % X4 = RL
put_value X2, A1
put_value X3, A2
put_value X4, A3
execute append/3
```

Klausel: `dx(F*G, (F*DG) + (G*DF))` ← `dx(F, DF), dx(G, DG)`
(Übersetzung von Unterstrukturen)

`F` und `DF` sind temporär, `G` und `DG` sind permanent.

Übersetzung:

```
allocate
get_structure '*'/2, A1
unify_variable X1        % X1 = F
unify_variable Y1        % Y1 = G
get_structure '+'/2, A2
unify_variable X3
unify_variable X4
get_structure '*'/2, X3
unify_value X1
unify_variable Y2        % Y2 = DG
get_structure '*'/2, X4
unify_value Y1
unify_variable X2        % X2 = DF
put_value X1, A1
put_value X2, A2
call dx/2, 2
put_value Y1, A1
put_value Y2, A2
deallocate
execute dx/2
```

An dem generierten Code sind noch zahlreiche Optimierungen möglich. Darauf werden wir später noch eingehen.

4.2.2 Indizierungsinstruktionen

Alle Klauseln für ein Prädikat werden übersetzt, indem die einzelnen Klauseln nach dem o.a. Schema übersetzt werden und zwischen die übersetzten Klauseln Befehle generiert werden, die die einzelnen Klauseln miteinander verbinden, so daß man im Falle eines Fehlschlages in einer Klausel sofort die nächste ausprobieren kann. Diese speziellen Befehle sind für die Verwaltung der Backtrackpunkte zuständig und realisieren außerdem ein einfaches Indizierungsschema zum schnellen Zugriff auf passende Klauseln.

Das **Indizierungsschema** basiert auf dem Wert des Hauptfunktors des 1. Arguments (Register `A1`) beim Aufruf, weil die Erfahrung gezeigt hat, daß in vielen Fällen allein durch diesen Wert genau eine passende Klausel bestimmt ist (vgl. z.B. `append`: Wenn das erste Argument eines `append`-Literals eine Liste ist, dann paßt zum Beweis des Literals nur genau eine Klausel, je nachdem ob die Liste leer oder nicht-leer ist). Falls dieses 1. Argument aber eine ungebundene Variable ist, dann muß natürlich jede Klausel ausprobiert werden. Daher werden auf jeden Fall alle Klauseln eines Prädikats sequentiell miteinander verbunden: Seien `K1, ..., Km` alle Klauseln für ein Prädikat `p/n`. Dann wird zunächst immer folgendes Schema generiert:

```
p/n: try_me_else L2, n
     < Code für Klausel K1>
L2:  retry_me_else L3
     < Code für Klausel K2>
L3:  retry_me_else L4
     ...
Lm:  trust_me_else fail
     < Code für Klausel Km>
```

Die neuen Befehle bedeuten:

Instruktion	Bedeutung
`try_me_else L, n`	Erzeuge einen neuen Backtrackpunkt auf dem lokalen Stack, in dem die aktuellen Werte der Register `A1, ..., An`, `CP`, `E`, `B`, `TR` und `H` in die entsprechenden Komponenten gesichert werden. Die Marke `L` ist die Adresse der nächsten Klausel und diese wird daher in der Komponente `BP` gesichert. `B` wird auf den neuen Backtrackpunkt gesetzt. Diese Anweisung wird unmittelbar vor der ersten Klausel eines Prädikats generiert (falls es mehr als eine Klausel gibt).
`retry_me_else L`	Diese Instruktion steht unmittelbar vor einer mittleren Klausel eines Prädikats. `L` ist die Adresse der nächsten Klausel. Die Instruktion bewirkt, daß die Komponente `BP` im letzten Backtrackpunkt auf `L` gesetzt wird: `B.BP := L`
`trust_me_else fail`	Diese Instruktion steht unmittelbar vor der letzten Klausel eines Prädikats. Der letzte Backtrackpunkt kann daher gelöscht werden: `B := B.BB`

Anmerkung: In der Originalbeschreibung der WAM [War83] wird der Parameter `n` in der Instruktion `try_me_else` nicht aufgeführt. Allerdings wird dort auch nicht erklärt, wie die Anzahl der zu sichernden Argumentregister in dieser Instruktion berechnet werden kann. Daher führen wir hier zu diesem Zweck den Zusatzparameter `n` ein. Alternativ könnte man den jeweils aktuellen Wert von `n` auch in einer globalen Variablen speichern, die in den `call`- und `execute`-Instruktionen gesetzt wird [AK91].

Die `try_me_else`-Kette von Instruktionen wird immer dann durchlaufen, wenn das 1. Argument beim Aufruf eine ungebundene Variable ist. In allen anderen Fällen kann man gezielt passende Klauseln herausfiltern. Z.B. muß zu einem `append`-Aufruf nur die 1. Klausel ausprobiert werden, wenn `A1` eine Konstante enthält, bzw. nur die 2. Klausel, wenn `A1` eine nicht-leere Liste enthält. Zu diesem Zweck gibt es folgende Befehle:

Instruktion	Bedeutung
`switch_on_term Lv, Lc, Ll, Ls`	Diese Instruktion leitet eine Folge von Klauseln ein, die im Klauselkopf keine Variable als erstes Argument haben. Der aktuelle Wert von `A1` wird dereferenziert, und je nach Typ des Arguments wird eine der 4 angegebenen Adressen angesprungen: Bei einer ungebundenen Variable `Lv`, bei einer Konstanten `Lc`, bei einer (nichtleeren) Liste `Ll` und bei einer Struktur `Ls`.
`switch_on_constant n, Table`	Dieser Befehl erlaubt einen schnellen Zugriff auf eine Gruppe von Klauseln, deren erstes Argument Konstanten sind. `Table` ist als Hash-Tabelle organisiert, die zu einer Konstanten die Adresse der zugehörigen Klausel(n) liefert. Wenn zu einer Konstanten kein Eintrag existiert, dann wird die Instruktion „`fail`“ ausgeführt. `n` ist die Größe der Hash-Tabelle (eine Potenz von 2).
`switch_on_structure n, Table`	Wie `switch_on_constant`, jedoch für Funktoren anstelle von Konstanten.
`try L, n`	Diese Instruktion ist die erste einer Folge von Instruktionen, die den Zugriff auf Klauseln organisieren, bei denen das erste Argument den gleichen Index (z.B. identische Konstanten) hat. Sie hat eine ähnliche Wirkung wie `try_me_else`: Ein Backtrackpunkt wird angelegt, die ersten `n` Argumentregister werden gesichert, ebenso die Register `CP`, `E`, `B`, `TR` und `H`. Die Komponente `BP` des neuen Backtrackpunkts wird auf die Anweisung gesetzt, die dieser `try`-Instruktion folgt. Anschließend wird `B` auf den neuen Backtrackpunkt gesetzt und der Programmzeiger `P` erhält den Wert `L`.
`retry L`	Diese Anweisung steht in der Mitte zwischen einer `try`- und `trust`-Instruktion. Die Alternativklausel im aktuellen Backtrackpunkt wird auf die Adresse hinter dieser Anweisung gesetzt, und `P` erhält den Wert `L`: `B.BP :=` <Adresse nach dieser Instruktion> `P := L`

`trust L`	Diese Instruktion ist die letzte einer `try-retry`-Kette. Sie löscht den letzten Backtrackpunkt:
	`B := B.BB`
	`P := L`

Die (`re`)`try_me_else`-Anweisungen dienen also zum Verbinden einzelner Klauseln, d.h. sie stehen unmittelbar *vor* der übersetzten Klausel, während die `switch`- und (`re`)`try`-Anweisungen ein Indizierungsschema *über* den Klauseln bilden, d.h. sie stehen zusammen vor einer Gruppe von Klauseln mit ähnlichen Kopfargumenten. Es folgt zunächst ein einfaches Beispiel für das Indizierungsschema:

```
p(a)    ← ...    % Klausel 1
p(b)    ← ...    % Klausel 2
p(f(X)) ← ...    % Klausel 3
p(a)    ← ...    % Klausel 4
p(g(X)) ← ...    % Klausel 5
```

```
p(a)    ← ...    % Klausel 6
```

Übersetzung mit Indizierungsschema:

```
p/1: switch_on_term L1, Lc1, fail, Ls1
Lc1: switch_on_constant 2, [a: Lc2,  b: L2a]
Ls1: switch_on_structure 2, [f/1: L3a,  g/1: L5a]
Lc2: try L1a, 1
     retry L4a
     trust L6a

L1:  try_me_else L2, 1
L1a: < Code für 1. Klausel>
L2:  retry_me_else L3
L2a: < Code für 2. Klausel>
L3:  retry_me_else L4
L3a: < Code für 3. Klausel>
L4:  retry_me_else L5
L4a: < Code für 4. Klausel>
L5:  retry_me_else L6
L5a: < Code für 5. Klausel>
L6:  trust_me_else fail
L6a: < Code für 6. Klausel>
```

Die `try-retry-trust`-Kette wird also nur dann aufgebaut, wenn es mehr als eine Klausel mit gleichem Schlüssel gibt. Die (`re`)`try_me_else`-Kette wird beim Backtracking durchlaufen, wenn das 1. Argument im aktuellen Aufruf eine ungebundene Variable ist. Wenn dies aber z.B. der Term `f(...)` ist, dann wird kein Backtrackpunkt aufgebaut und nur Klausel 3 ausprobiert.

Das nachfolgende, etwas komplexere Beispiel ist aus [War83] entnommen. Es zeigt, daß es nicht nur möglich ist, alle Klauseln eines Prädikats mit `switch_on_term` zu indizieren, sondern die Indizierung kann auch in kleineren Gruppen erfolgen, falls in der Mitte eine Klausel mit variablem 1. Argument im Kopf vorkommt.

```
call(X or Y)  ← call(X)
call(X or Y)  ← call(Y)
call(trace)   ← trace
call(notrace) ← notrace
call(nl)      ← nl
call(X)       ← builtin(X)
call(X)       ← ext(X)
call(call(X)) ← call(X)
call(repeat)  ←
call(repeat)  ← call(repeat)
call(true)    ←
```

Übersetzung:

```
call/1: try_me_else C6a, 1
        switch_on_term C1a, L1, fail, L2
L1:     switch_on_constant 4, [trace: C3,  notrace: C4,  fail,  nl: C5]
L2:     switch_on_structure 1, [or/2: L3]

L3:     try C1, 1
        trust C2

C1a:    try_me_else C2a, 1              % call(
C1:     get_structure or/2, A1          % or(
        unify_variable A1               % X,
        unify_void 1                    % Y)) ←
        execute call/1                  % call(X)

C2a:    retry_me else C3a               % call(
C2:     get_structure or/2, A1          % or(
        unify_void 1                    % X,
        unify_variable A1               % Y)) ←
        execute call/1                  % call(Y)

C3a:    retry_me_else C4a               % call(
C3:     get_constant trace, A1          % trace) ←
        execute trace/0                 % trace

C4a:    retry_me_else C5a               % call(
C4:     get_constant notrace, A1        % notrace) ←
        execute notrace/0               % notrace

C5a:    trust_me_else fail              % call(
C5:     get_constant nl, A1             % nl) ←
        execute nl/0                    % nl

C6a:    retry_me_else C7a               % call(X) ←
        execute builtin/1               % builtin(X)

C7a:    retry_me_else L4                % call(X) ←
        execute ext/1                   % ext(X)

L4:     trust_me_else fail
        switch_on_term C8a, L5, fail, L7

L5:     switch_on_constant 2, [repeat: L6, true: C11]


L6:     try C9, 1
        trust C10

L7:     switch_on_structure 1, [call/1: C8]
```

```
C8a:    try_me_else C9a, 1          % call(
C8:     get_structure call/1, A1    % call(
        unify_variable A1           % X)) ←
        execute call/1              % call(X)

C9a:    retry_me_else C10a          % call(
C9:     get_constant repeat, A1     % repeat
        proceed                     % ) ←

C10a:   retry_me_else C11a          % call(
C10:    get_constant repeat, A1     % repeat) ←
        put_constant repeat, A1     % call(repeat
        execute call/1              % )

C11a:   trust_me_else fail          % call(
C11:    get_constant true, A1       % true
        proceed                     % ) ←
```

In diesem Beispiel wird zunächst eine äußere (`re`)`try_me_else`-Kette für die Klauselgruppen 1.-5. Klausel, 6. Klausel, 7. Klausel und 8.-11. Klausel aufgebaut und dann jeweils ein Indizierungsschema für die 1.-5. Klausel und für die 8.-11. Klausel. Ein durchgängiges Indizierungsschema, das alle Klauseln umfaßt, ist nicht möglich, weil die 6. und 7. Klausel variable Kopfargumente haben. In diesem Beispiel ist es möglich, daß ein Literal ein oder zwei Backtrackpunkte hat. Welcher der Fälle eintritt, ist statisch nicht zu entscheiden, sondern dies hängt vom Argument des aktuellen `call`-Literals ab.

Wir schließen die Beschreibung der WAM-Instruktion mit zwei Instruktionen ab, die im übersetzten Programm nicht explizit stehen, sondern nur implizit bei der Ausführung anderer Instruktionen aufgerufen werden:

Instruktion	**Bedeutung**
`fail`	Diese Instruktion wird ausgeführt, wenn während einer Unifikation ein Fehlschlag auftritt. Sie realisiert einen Backtracking-Schritt: Alle Variablen auf dem Trail oberhalb von `B.TR` werden auf „ungebunden“ gesetzt, die `A`-Register und die Register `CP`, `E`, `TR`, `H` werden auf die alten, im aktuellen Backtrack-Punkt gespeicherten Werte gesetzt und `P` wird auf die nächste alternative Klausel (= `B.BP`) gesetzt.
`trail(R)`	Diese Instruktion wird immer dann ausgeführt, wenn eine Variable während der Unifikation gebunden wird. Wenn die Variable `R` auf dem Heap vor dem letzten Backtrackpunkt erzeugt wurde (d.h. sie steht vor `B.BH`), oder wenn sie auf dem Stack vor dem letzten Backtrackpunkt steht (d.h. sie steht vor `B`), dann wird `R` auf den Trail geschrieben. Ansonsten wird nichts gemacht.

Damit sind alle Instruktionen der WAM beschrieben. Dies sind die Basis-Instruktionen, die zu jeder Implementierung gehören. Bei realen Implementierungen wird dieser Instruktionssatz noch um

zusätzliche Befehle erweitert, die einerseits Spezialisierungen dieser Befehle sind (z.B. spezielle `unify`-Instruktionen für den Schreib-Modus) oder andererseits auch zur Implementierung von nicht-logischen Konstrukten dienen. Wir werden darauf noch eingehen.

Die WAM und die Übersetzung von logischen Programmen in WAM-Code wurde hier nur umgangssprachlich beschrieben. Eine ausführliche formale Spezifikation dieser Aspekte findet man in [Pro87]. Ein Überblick hierüber ist in [Han88] angegeben. Eine genaue Spezifikation der WAM-Instruktionen befindet sich außerdem in [AK91]. Der eigentliche Compiler zur Übersetzung von logischen Programmen in WAM-Code kann ein Programm sequentiell übersetzen: Jedem Quellsprachkonstrukt entspricht eine WAM-Instruktion. Zur konkreten Übersetzung werden jedoch viele Hilfsinformationen benötigt wie Klassifizierung und Numerierung der Variablen, Struktur der Klauselköpfe (Indizierungsschema) etc. Insofern hat der Compiler zwei Phasen: eine Analysephase zur Berechnung der Hilfsinformationen und eine Codegenerierungsphase, bei der der WAM-Code ausgegeben wird. Beschreibungen solcher Compiler befinden sich in [Pro87] (formal) und [VR84].

Implementierung der WAM

Die WAM-Instruktionen sind keine Befehle einer bestimmten Hardware-Architektur, sondern sie müssen erst noch auf einer realen Hardware implementiert werden. Wie zu Beginn dieses Kapitels erwähnt, gibt es im wesentlichen drei Möglichkeiten zur Implementierung: Emulation, Übersetzung in Maschinencode oder Konstruktion einer speziellen Hardware. Wir wollen uns im weiteren nicht mit der konkreten Implementierung der WAM beschäftigen und geben deshalb an dieser Stelle einige Hinweise. Bei der *Emulation* müssen die WAM-Instruktionen in einem möglichst kompakten Bytecode dargestellt werden, der von dem Emulatorprogramm schnell decodiert werden kann. Vorschläge für ein mögliches Datenformat findet man in [War83].

Bei der *Übersetzung in Maschinencode* muß man sich die Verteilung der WAM-Register auf die zur Verfügung stehenden Maschinenregister sorgfältig überlegen. Dann kann man einige WAM-Instruktionen direkt in einzelne Maschinenbefehle übersetzen (`proceed`, `execute`, `deallocate` etc.), andere Instruktionen sind so komplex, daß sich die Übersetzung in einen Unterprogrammaufruf empfiehlt (`fail`, `switch_on_constant` etc.) Hinweise für eine solche Implementierung findet man in [Pro87]. Das ProCom-Prolog-System übersetzt WAM-Programme in 68020-Maschinen-Programme.

Der Vorteil der Emulation ist die kompakte Größe der übersetzten Programme und die einfache Implementierung des Emulators. Der Vorteil der Übersetzung in Maschinencode ist die höhere Geschwindigkeit (da kein Emulator notwendig ist) und die Möglichkeit, das Maschinenprogramm durch Peephole-Optimierungen noch zu beschleunigen (vgl. z.B. [ASU85]). Der Nachteil dieser Methode ist allerdings die Codegröße: Das übersetzte Maschinenprogramm kann bei größeren Anwendungen schnell so groß werden, daß es nicht mehr in den Hauptspeicher paßt und daher das Paging beginnt. Da die Daten häufig nicht-lokal sind, kann sich das dann sehr negativ bemerkbar machen.

Eine gute Lösung wäre die Kombination beider Methoden, was jedoch zur Erhaltung der Konsistenz einen höheren Verwaltungsaufwand erfordert. Eventuell können auch hier Techniken der partiellen Auswertung weiterhelfen.

4.3 Eigenschaften der WAM

In diesem Kapitel wollen wir zum besseren Verständnis der Arbeitsweise der WAM einige Eigenschaften dieser Maschine genauer betrachten.

4.3.1 Anlegen von Umgebungen und Backtrackpunkten

Wenn ein Literal bewiesen wird, so werden beim Beweis je nachdem, welche Klauseln hierfür vorhanden sind, keine oder eine Umgebung und kein, ein oder auch zwei(!) Backtrackpunkte angelegt. Hierzu einige Beispiele, wobei wir immer alle definierenden Klauseln für das Prädikat angeben:

1. $L \leftarrow$
   (d.h. es gibt nur ein Faktum für das Prädikat) In diesem Fall wird weder eine Umgebung noch ein Backtrackpunkt beim Beweis erzeugt.

2. $L \leftarrow L_1, L_2, L_3$
   (es gibt nur eine Klausel mit 3 Rumpfliteralen) In diesem Fall wird eine Umgebung (für die permanenten Variablen und die Rücksprungadresse `CP`), aber kein Backtrackpunkt angelegt.

3. `p(X)` $\leftarrow$
   `p(Y)` $\leftarrow$

   Übersetzung in WAM-Code:

   ```
   p/1: try_me_else P2, 1
        get_variable X1, A1
        proceed
   P2:  trust_me_else fail
        get_variable X1, A1
        proceed
   ```

   In diesem Fall wird keine Umgebung, aber immer ein Backtrackpunkt angelegt.

4. `p(a)` $\leftarrow$
   `p(b)` $\leftarrow$

   Es wird keine Umgebung angelegt. Das Anlegen eines Backtrackpunkts entscheidet sich erst zur Laufzeit: Wenn beim Aufruf `p(t)` das aktuelle Argument `t` eine ungebundene Variable ist, dann wird ein Backtrackpunkt angelegt, ansonsten wird aufgrund des Indizierungsschemas keiner angelegt, da es dann nur höchstens eine passende Klausel gibt.

5. `p(a)` $\leftarrow L_1, L_2$
   `p(b)` $\leftarrow$

   Wenn die erste Klausel ausprobiert wird, dann wird eine Umgebung angelegt. Das Anlegen eines Backtrackpunkts entscheidet sich dynamisch (vgl. 4.)

6. `p(a)` $\leftarrow \cdots$
   `p(X)` $\leftarrow \cdots$
   `p(b)` $\leftarrow \cdots$ $(*)$
   `p(b)` $\leftarrow \cdots$
   `p(Y)` $\leftarrow \cdots$

   Durch das Indizierungsschema wird zunächst ein Backtrackpunkt für die äußere try_me_else-Kette angelegt, die die Klauseln 1, 2, 3 und 5 verbindet. Falls die 3. Klausel angewendet wird, wird vor Anwendung ein weiterer Backtrackpunkt angelegt, d.h. an der Stelle $(*)$ existieren zwei Backtrackpunkte für das zu beweisende `p`-Literal.

Da also zum Beweis eines Literals kein, ein oder auch zwei Backtrackpunkte angelegt werden können und zudem die genaue Anzahl erst dynamisch festgelegt wird, weiß man zur Laufzeit nie, zu welchem Ziel der letzte Backtrackpunkt gehört. Im Gegensatz zum einfachen deterministischen Interpreter, der die SLD-Resolution implementiert (vgl. Kapitel 2), gehört hier der letzte Backtrackpunkt nicht notwendigerweise zum Vaterziel (im SLD-Baum).

4.3.2 Verzögerte Prozeduraufrufe

Die WAM versucht, im Gegensatz zu Implementierungen imperativer Programmiersprachen, das Anlegen von Speicherplatz möglichst lange zu verzögern, so daß es eventuell überflüssig wird. Besonders deutlich sieht man dies beim Aufrufmechanismus für Klauseln. Prozeduraufrufe bei imperativen Programmiersprachen werden üblicherweise dadurch realisiert, daß die Rückkehradresse (= Adresse der nächsten Anweisung hinter dem Prozeduraufruf) auf dem Stack gelegt wird und dann die Prozedur angesprungen wird. Prozeduren entsprechen in logischen Programmen den Klauseln, und ein Prozeduraufruf entspricht dem Beweis eines Literals. In der WAM wird beim Beweis eines Literals (i.a. eingeleitet durch die WAM-Instruktion „`call P, n`“) die Rückkehradresse jedoch nicht sofort auf den Stack gelegt (was ja Speicherplatz kostet), sondern sie wird zunächst in das Register `CP` geschrieben und dann wird der Code für das Prädikat angesprungen. Falls nun die nächste anzuwendende Klausel komplex ist (d.h. sie enthält mehr als ein Literal im Rumpf), dann wird auf dem Stack eine neue Umgebung angelegt und die Rückkehradresse (Register `CP`) wird auf dem Stack in der neuen Umgebung gesichert. In allen anderen Fällen wird die Rückkehradresse nicht auf dem Stack gespeichert. Durch diese Technik werden in vielen Fällen Literale nur durch Sprünge und nicht durch teurere Unterprogrammaufrufe bewiesen.

4.3.3 Verkürzung von Umgebungen: „environment trimming“

Wie schon erwähnt, wird während der Abarbeitung einer Klausel die Umgebung unter bestimmten Umständen verkürzt, indem nicht mehr notwendige permanente Variablen gelöscht werden. Darauf gehen wir noch kurz genauer ein.

Beispiel: Abarbeitung der Klausel

`p(X)` $\leftarrow$ `q(Y), r(X), s(Y)`

Permanente Variablen in der Umgebung dieser Klausel:

```
Y1 = Y
Y2 = X
```

Erzeugter WAM-Code:

```
allocate
get_variable Y2, A1
put_variable Y1, A1
call q/1, 2
put_value Y2, A1
call r/1, 1
put_unsafe_value Y1, A1
deallocate
execute s/1
```

Bei Aufruf von `q/1` wird noch die gesamte Umgebung benötigt (2. Argument = 2), während bei Aufruf von `r/1` nur noch die Variable `Y1` (=`Y`) in der Umgebung benötigt wird (2. Argument = 1). Wenn also beim Beweis des Literals `r(X)` eine Umgebung oder ein Backtrackpunkt angelegt wird, so gibt es zwei Möglichkeiten:

1. Wurde beim Beweis von `q(Y)` ein Backtrackpunkt angelegt und ist dieser noch vorhanden, dann liegt er oberhalb der Umgebung für diese Klausel auf dem Stack, d.h. eine neue Umgebung bzw. Backtrackpunkt wird oberhalb des Backtrackpunkts für `q(Y)` angelegt. Die Umgebung bleibt somit unverändert. Backtrackpunkte frieren somit die Länge der darunter liegenden Umgebungen ein!

2. Wurde beim Beweis von `q(Y)` kein Backtrackpunkt angelegt oder ist ein evtl. angelegter Backtrackpunkt gelöscht worden, dann ist die aktuelle Umgebung das oberste Objekt auf dem Stack. Da aber die Variable `Y2` in der Umgebung nicht mehr gebraucht wird, kann eine neue Umgebung oder ein neuer Backtrackpunkt an der Stelle beginnen, an der `Y2` gespeichert ist. Da bei Abarbeitung von `r/1` `CP` auf die Anweisung nach dem `call` verweist, kann man über `CP` auf das 2. Argument des `call`-Aufrufs zugreifen und daraus die Stelle berechnen, an der ein neues Objekt auf dem Stack gespeichert werden kann. In diesem Fall wird also die Länge der aktuellen Umgebung verkürzt.

4.3.4 „Tail recursion optimization“

Vor dem Aufruf des letzten Literals im Rumpf einer Klausel mit mehreren Literalen wird die aktuelle Umgebung mit `deallocate` freigegeben, was einer Verkürzung der Umgebung auf 0 Variablen entspricht. Dies hat ein Löschen der Umgebung auf dem Stack nur zur Folge, wenn oberhalb dieser Umgebung keine Backtrackpunkte stehen (vgl. „environment trimming“). Bei deterministischen Berechnungen wird für den letzten Aufruf im Rumpf also kein Speicherplatz benötigt und der Aufruf

erfolgt auch nicht als Unterprogrammaufruf (`call`), sondern als Sprung (`execute`). Somit wird die rekursive Definition von `append` genauso effizient wie eine Implementierung mit einer while-Schleife abgearbeitet (vorausgesetzt, daß bei Aufruf das 1. Argument ein Grundterm ist). Dieser Mechanismus ist eine Verallgemeinerung des „tail recursion optimization“ (Umwandlung von Rekursion am Ende einer Prozedur in eine Schleife) und wird daher auch als „**last goal optimization**“ bezeichnet.

4.4 Optimierungsmöglichkeiten

An dem durch unser Übersetzungsschema erzeugten WAM-Code können noch viele Optimierungen vorgenommen werden. Einige Optimierungsmöglichkeiten auf der Ebene der WAM-Instruktionen geben wir im folgenden an (zusätzlich können bei Übersetzung des WAM-Codes in Maschinencode noch Peephole-Optimierungen vorgenommen werden).

4.4.1 Identifikation von Registern

Beobachtung: Register für temporäre Variablen (`Xi`) und Argumente (`Ai`) haben eine ähnliche Funktion. Daher kann man sie identifizieren [War83]: Im folgenden soll also grundsätzlich `Xi`=`Ai` gelten. Die ersten Register werden für Argumente benutzt, und die weiteren `A`-Register für temporäre Variablen. Der Compiler kann durch geschickte Verteilung der temporären Variablen auf die `A`-Register einen verbesserten Code erzeugen, wenn folgende Optimierungen berücksichtigt werden:

- Die Instruktionsfolge

  ```
  get_variable Xj, Ai
  put_value Xj, Ai
  ```

  kann gelöscht werden, wobei jedes weitere nachfolgende Vorkommen von `Xj` durch `Ai` ersetzt wird.

- In den Instruktionen kann überall, wo `Xi` steht, auch `Ai` vorkommen.

- Die Instruktionen

  ```
  get_variable Xi, Ai
  ```

  und

  ```
  put_value Xi, Ai
  ```

  sind überflüssig (dies sollte der Compiler bei der Verteilung der temporären Variablen auf die Register `Ai` beachten!).

- Die Instruktionsfolge

  ```
  get_variable Xi, Ai
  get_value Xi, Aj
  ```

  kann durch

  ```
  get_value Ai, Aj
  ```

  ersetzt werden.

Unter Berücksichtigung dieser Optimierungen kann für das Programm

```
append([], L, L) ←
append([E|R], L, [E|RL]) ← append(R, L, RL)
```

der folgende WAM-Code erzeugt werden:

```
append/3: switch_on_term L1, L1c, L2l, fail
L1:       try_me_else L2, 3
L1c:      get_nil A1
          get_value A2, A3
          proceed
L2:       trust_me_else fail
L2l:      get_list A1
          unify_variable X4    % X4 = E
          unify_variable A1    % A1  kann überschrieben werden
          get_list A3
          unify_value X4
          unify_variable A3    % A3  kann überschrieben werden
          execute append/3
```

Gegenüber der ersten Übersetzung von `append/3` sind hier 5 Instruktionen eingespart worden. Wir werden später sehen, wie der Compiler eine solche verbesserte Zuordnung von temporären Variablen zu Registern automatisch finden kann.

4.4.2 Verbesserung des Indizierungsschemas

Bisher wird nur indiziert über das erste Argument des zu beweisenden Literals. Je nach Form der Klauselköpfe kann die Indizierung über ein anderes Argument günstiger sein (wenn z.B. das erste Argument in den Klauselköpfen immer variabel ist). Daher könnte man für jedes Prädikat ein anderes Indizierungsargument angeben, d.h. die `switch_on`-Anweisungen müssen so erweitert werden, daß sie das entsprechende Argumentregister als Parameter haben. Z.B. kann der Benutzer beim BIM-Prolog-System die Argumentnummer, über die indiziert werden soll, explizit angeben. Diese Idee kann man fortführen und nicht nur über beliebige Argumente indizieren, sondern über mehrere Argumente gleichzeitig und auch über komplexere Terme in den Argumenten. Ideen hierzu findet man in [WP84].

Das Anlegen von Backtrackpunkten ist eine aufwendige Operation und benötigt viel Speicherplatz. Daher sollte man bei der Implementierung immer bestrebt sein, dies nach Möglichkeit zu vermeiden. Bei dem o.a. Indizierungsschema werden in manchen Fällen sogar zwei Backtrackpunkte vor Ausführung einer Klausel angelegt. Dies kann man in vielen Fällen vermeiden durch ein modifiziertes Indizierungsschema. Vorschläge hierzu findet man in [VR84], [BBCT86], [VRDW87] und [Car87]. Eine andere Alternative zur Einsparung von Backtrackpunkten findet man in [Tur86]: Wenn aufgrund des Indizierungsschemas zwei Backtrackpunkte angelegt werden, dann unterscheiden sich diese nur in den Komponenten `BB` (Verkettung der Backtrackpunkte) und `BP` (nächste Alternative). Aus diesem Grund kann man die beiden Punkte zu einem verschmelzen, der aber dann eine zweite `BP` Komponente enthalten muß. Das erste `try_me_else` legt dann einen solchen Backtrackpunkt an, während ein weiteres `try(_me_else)` nur noch eine weitere Komponente `BP` speichert.

4.4.3 Auscompilieren des Lese-/Schreibmodus

Die `unify`-Instruktion zur Unifikation von Komponenten einer Struktur arbeiten grundsätzlich in zwei Modi, d.h. sie müssen immer das `RW`-Register abfragen. Strukturen im Rumpf einer Klausel werden jedoch infolge der vorhergehenden `put`-Instruktion immer im Schreibmodus bearbeitet. Daher kann man spezielle `unify_write`-Instruktionen definieren, die wie `unify`-Instruktionen arbeiten, jedoch das `RW`-Register nicht abfragen und grundsätzlich im Schreibmodus arbeiten. Strukturen im Rumpf einer Klausel werden dann in `put_structure`- bzw. `put_list`-Anweisungen und `unify_write`-Anweisungen übersetzt.

Turk [Tur86] geht noch einen Schritt weiter und schlägt vor, auch im Klauselkopf nur noch spezielle `unify_read`- und `unify_write`-Anweisungen zu verwenden, so daß das Register `RW` überhaupt nicht mehr abgefragt werden muß: Dazu wird (zur Laufzeit) zunächst geprüft, ob das jeweilige Argument eine ungebundene Variable ist. Falls ja, dann wird die `unify_write`-Codesequenz ausgeführt, sonst die `unify_read`-Codesequenz. Der Nachteil dieses Ansatzes ist der größere WAM-Code.

4.4.4 Verbesserte (lokale) Registerallokation

Alle Variablen in einer Klausel werden entweder in Registern (`Xi = Ai`) (temporäre Variablen) oder auf dem Stack gespeichert (permanente Variablen). Da es innerhalb einer Klausel keine Einschränkungen über die Verwendung von Registern gibt, können Variablen, die in mehr als einem Rumpfliteral vorkommen, nicht in Registern gehalten werden, sie sind daher als permanent klassifiziert.

Der erzeugte Code für eine Klausel kann verbessert werden (bzgl. Ausführungsgeschwindigkeit und Speicherplatzbedarf), indem

- mehr Klauselvariablen als temporär klassifiziert werden,
- die temporären Variablen so auf Register verteilt werden, daß die Anzahl der Datenbewegungen zwischen Registern verkleinert wird.

Einen Vorschlag für solche Verbesserungen findet man in [Deb86]. Diesen wollen wir im folgenden kurz darstellen.

Zunächst wird der Begriff der temporären Variable erweitert:

Def.: Ein Prädikat heißt **primitiv** („**in-line predicate**“), wenn es direkt durch eine Folge von WAM-Instruktionen ausgeführt werden kann (ohne einen Prozeduraufruf).

Primitive Prädikate sind z.B. Typtests für Terme, arithmetische Berechnungen, Unifikation etc. Bei primitiven Prädikaten ist der Bedarf an Registern zur Compilezeit bekannt, während andere Prädikate potentiell beliebig viele Register verändern können. Daher bleibt der Wert von temporären Variablen beim Aufruf primitiver Prädikate erhalten und somit definieren wir:

Def.: Ein **Block** („**chunk**“) ist eine (evtl. leere) Folge von primitiven Prädikaten, gefolgt von einem Prädikat, das nicht primitiv ist.

Blöcke sind also die längsten Sequenzen in einem Klauselrumpf, in denen zur Compilezeit festgestellt werden kann, welche Register verändert werden und welche nicht. Somit brauchen Variablen nur permanent sein, wenn sie in zwei verschiedenen Blöcken auftreten:

Def.: Eine Variable in einer Klausel ist **temporär**, wenn gilt:

1. Sie tritt zum ersten Mal im Kopf, in einer Struktur oder im letzten Block des Rumpfes auf.
2. Die Variable kommt nicht in zwei verschiedenen Blöcken des Rumpfes vor.
3. Wenn die Variable im Kopf vorkommt, dann kommt sie höchstens noch im ersten Block des Rumpfes vor.

Eine Variable ist **permanent**, wenn sie nicht temporär ist.

Mit dieser Neudefinition werden nun weniger Variablen als permanent klassifiziert:

```
p(X, Y) ←   var(X), Y =.. [a|_], q(X, Y, Z), r(Z)
```

Hier bilden die ersten drei Rumpfliterale einen Block. Somit ist nur `Z` permanent.

Da am Ende eines Blocks ein nicht-primitives Prädikat ausgeführt wird, weiß man nach einem Block nichts über den Inhalt der Register. Daher werden wir im folgenden nur noch die *Codeerzeugung für einzelne Blöcke* betrachten.

Grundsätzlich setzen wir im folgenden voraus, daß die **A- und X-Register identisch** sind. Die Instruktion `get_variable Xi, Aj` bzw. `put_value Xi, Aj` entsprechen somit Transferoperationen zwischen Registern (kopiere Inhalt von Register `Aj` nach `Xi` bzw. kopiere Inhalt von Register `Xi` nach `Aj`). Wesentliches Ziel ist es, die Anzahl dieser Instruktionen durch eine geschickte Zuordnung von Registern zu vermindern.

Zu diesem Zweck erfolgt die Codeerzeugung in zwei Phasen:

1. Analyse des Quellprogramms (hier: eine Klausel bzw. ein Block)
2. Sequentielle Erzeugung des Codes, wobei nun für temporäre Variablen bestimmte Register reserviert werden müssen.

Im wesentlichen geht es im folgenden um eine geschickte Lösung des Problems, wie die temporären Variablen auf die vorhandenen Register verteilt werden. Wir geben dazu verschiedene Implementierungen der beiden Codeerzeugungsphasen an.

Naive Registerzuteilungsstrategie:

1. Markiere im Block für jede Variable das letzte Vorkommen dieser Variablen.
2. Verwalte während der Codeerzeugung eine Menge **INUSE** von Registern, die zur Zeit benutzt werden (Register, die nicht in INUSE vorkommen, heißen **freie Register**).

a) Zuteilung eines neuen Registers:
   - Wähle ein Register R mit $R \notin INUSE$, d.h. R ist frei.
   - $INUSE := INUSE \cup \{R\}$

b) Nach Codeerzeugung für eine temporäre Variable, die in Register R ist und danach im Code nicht mehr vorkommt:
   - $INUSE := INUSE - \{R\}$

Konvention bei diesen und allen nachfolgenden Algorithmen: Wenn bei einer Registerauswahl mehrere mögliche Register zur Verfügung stehen, dann wähle zuerst das mit der kleinsten Nummer.

Nachteil des Algorithmus': Es werden i.a. zu viele Registertransferoperationen erzeugt, da nicht beachtet wird, wo eine temporäre Variable später noch gebraucht wird. Daher:

Verbesserung der naiven Registerzuteilungsstrategie:

1. Berechne zusätzlich für jede temporäre Variable T die folgenden Mengen:

   USE(T): Menge der Argumentpositionen im letzten Literal des Blocks (und im Klauselkopf, falls der Block der erste im Rumpf ist), bei denen T als Argument steht.

   NOUSE(T): Menge der Argumentpositionen p im letzten Literal des Blocks, die als Argument eine temporäre Variable U haben mit $U \neq T$ und $p \notin USE(T)$

   (Wichtig sind nur die Argumente im letzten Literal eines Blocks, da für die anderen primitiven Literale die Register beliebig verteilt werden können.)

2. Codeerzeugung: wie bisher, aber bei Zuteilung eines Registers R für eine temporäre Variable T:
   - Wähle ein *freies* Register R mit $R \in USE(T)$
   - Falls dies nicht möglich ist: Wähle ein *freies* Register R mit $R \notin NOUSE(T)$

Beispiel: `p(X, f(X), Y) ← X = [a|W], q(W, Y, X)`

Für diesen Block gilt:

$USE(\texttt{X}) = \{1,3\}$	$NOUSE(\texttt{X}) = \{2\}$
$USE(\texttt{Y}) = \{2,3\}$	$NOUSE(\texttt{Y}) = \{1\}$
$USE(\texttt{W}) = \{1\}$	$NOUSE(\texttt{W}) = \{2,3\}$

2. Klausel von append:

`append([E|R], L, [E|RL]) ← append(R, L, RL)`

Hierfür gilt:

$USE(\texttt{E}) = \emptyset$	$NOUSE(\texttt{E}) = \{1,2,3\}$
$USE(\texttt{R}) = \{1\}$	$NOUSE(\texttt{R}) = \{2,3\}$
$USE(\texttt{L}) = \{2\}$	$NOUSE(\texttt{L}) = \{1,3\}$
$USE(\texttt{RL}) = \{3\}$	$NOUSE(\texttt{RL}) = \{1,2\}$

Registerzuteilung mit verbesserter Strategie:

E → 4
R → 1
L → 2
RL → 3

Erzeugter Code für diese Klausel:

```
get_list A1
unify_variable X4
unify_variable A1
get_list A3
unify_value X4
unify_variable A3
execute append/3
```

Bei der bisherigen Strategie haben wir angenommen, daß eine temporäre Variable in einem Register während ihrer ganzen Lebenszeit gespeichert werden kann. Hierbei kann es jedoch zu Konflikten kommen, wenn das Register, in dem die temporäre Variable gespeichert ist, als Argument für einen Prädikataufruf benötigt wird:

```
p(X) ← q(a, X)
```

Hier ist $USE(\texttt{X}) = \{1, 2\}$, d.h. `X` kann zunächst in Register 1 gespeichert bleiben. Wenn jedoch Code für den Rumpf erzeugt wird, muß in Register 1 die Konstante `a` geladen werden, wodurch der Wert von `X` verloren gehen würde. Wir haben hier also einen Konflikt in der Registerzuteilung. Zur Lösung solcher Konflikte gibt es zwei prinzipielle Strategien:

Konfliktvermeidung: Es werden alle potentiellen Konflikte vorher berechnet und die Registerverteilung wird so geregelt, daß keine Konflikte entstehen können.

Konfliktlösung: Es wird im vorhinein nicht versucht, Konflikte zu vermeiden. Erst in dem Moment, in dem ein Konflikt auftritt, wird er aufgelöst.

Wir stellen im folgenden je eine Strategie zur Konfliktvermeidung, zur Konfliktlösung und schließlich eine Mischung aus beiden vor.

Registerzuteilungsstrategie mit Konfliktvermeidung:

1. Berechne für jede temporäre Variable T die Mengen $USE(T)$, $NOUSE(T)$ und

   CONFLICT(T): Menge der Argumentpositionen im letzten Literal des Blocks, bei denen nicht T steht, falls T überhaupt in diesem Literal vorkommt, ansonsten ist diese Menge leer.

2. Die Registerzuteilung für eine temporäre Variable T wird wie folgt geändert:

   - Wähle ein *freies* Register $R \in USE(T) - CONFLICT(T)$

- Falls dies nicht möglich ist:
  Wähle ein *freies* Register R mit $R \notin NOUSE(T) \cup CONFLICT(T)$

Vorteil dieser Strategie: Einfachheit

Nachteil dieser Strategie: Konflikte werden zu früh gelöst, was unnötige Operationen zur Folge hat

Beispiel: `p(X, Y, Z, a) ← q(Z, X, Y)`

Hierfür gilt:

$CONFLICT(\texttt{X}) = \{1, 3\}$
$CONFLICT(\texttt{Y}) = \{1, 2\}$
$CONFLICT(\texttt{Z}) = \{2, 3\}$

Erzeugter Code mit Konfliktvermeidung:

```
get_variable X5, A1
get_variable X6, A2
get_variable A1, A3
get_constant a, A4
put_value X5, A2
put_value X6, A3
execute q/3
```

Falls zur Laufzeit die Unifikation von `A4` mit der Konstanten `a` fehlschlägt, sind drei unnötige Registertransferoperationen ausgeführt worden.

Registerzuteilungsstrategie mit Konfliktlösung:

1. Berechne für jede temporäre Variable T die Mengen $USE(T)$ und $NOUSE(T)$.
2. Verwalte während der Codeerzeugung die Tabellen

   CONTENTS: Register → Temporäre Variablen ∪ {notemp}

   REGISTER: Temporäre Variablen → Register ∪ {noreg}

   $CONTENTS(R)$ gibt den jeweiligen Inhalt von Register R an, während $REGISTER(T)$ angibt, in welchem Register der Wert der temporären Variablen T gegenwärtig gespeichert ist.

   Die Codeerzeugung für einen Block (einschließlich des Klauselkopfes) wird bis auf das letzte Literal im Block genauso durchgeführt wie bei der verbesserten naiven Registerzuteilungsstrategie (Zuteilung mit Hilfe der USE- und $NOUSE$-Mengen), wobei die Tabellen $CONTENTS$ und $REGISTER$ entsprechend verwaltet werden.

   Wenn für das letzte Literal $p(t_1, \ldots, t_n)$ im Block für das Argument mit der Nummer k Code erzeugt werden soll, aber $CONTENTS(k) = T \neq notemp$ ist mit $T \neq t_k$, dann ist ein Konflikt vorhanden. Dieser wird wie folgt gelöst: Wähle ein Register R mit $R \notin NOUSE(T) \cup INUSE \cup \{k\}$ und erzeuge Code zum Laden des Registers R mit dem Inhalt von Register k. Dann:

$CONTENTS(R) := T$
$CONTENTS(k) := notemp$
$REGISTER(T) := R$

Anschließend wird der Code zum Laden des Arguments t_k in das Register k erzeugt.

Beispiel: `p(X, Y, Z, a) ← q(Z, X, Y)`

Erzeugter Code mit Konfliktlösung (Konflikte treten erst bei der Codeerzeugung für das Literal im Rumpf auf):

```
get_constant a, A4
get_variable X4, A1
put_value A3, A1
put_value A2, A3
put_value X4, A2
execute q/3
```

Der erzeugte Code ist besser als bei Konfliktvermeidung.

Der erzeugte Code kann aber in bestimmten Fällen auch wesentlich schlechter sein als bei der Konfliktvermeidung:

Beispiel: `p(X, a, b) ← q(c, d, e, f(X))`

Hier ist $USE($`X`$) = \{1\}$ und $NOUSE($`X`$) = \emptyset$. Wenn für das Rumpfliteral Code erzeugt wird, wird `X` aufgrund der Konfliktlösungsstrategie zunächst von Register 1 in Register 2 verschoben, dann von Register 2 in Register 3 usw.

Daher ist eine Mischung beider Strategien sinnvoller. Zunächst wird die Konfliktlösungsstrategie verwendet („Lasse die temporären Variablen so lange wie möglich in ihren Registern“) und falls ein Konflikt eintritt, wird er gelöst nach der Konfliktvermeidungsstrategie („Verschiebe die temporäre Variable in ein Register, das später keinen Konflikt ergibt“). Beachte, daß Konflikte nur entstehen können bei temporären Variablen, die im Klauselkopf als Argumente vorkommen.

Registerzuteilungsstrategie mit Konfliktlösung und -vermeidung:

1. Berechne für jede temporäre Variable T die Mengen $USE(T)$, $NOUSE(T)$ und $CONFLICT(T)$
2. Verwalte während der Codeerzeugung die Tabellen $CONTENTS$ und $REGISTER$. Registerzuteilung für eine temporäre Variable T:
   - Falls T ein Argument des Klauselkopfes ist, dann teile Register nach Konfliktlösungsstrategie zu (d.h. belasse T im Argumentregister),
   - sonst: teile Register nach Konfliktvermeidungsstrategie zu.

Die Codeerzeugung wird wie bei der Konfliktlösungsstrategie durchgeführt (aber mit obiger Registerzuteilung), jedoch wird, falls ein Konflikt bei der Codeerzeugung für das Argument t_k des letzten Literals im Block auftritt, wie folgt verfahren: Wähle ein Register R nach der Konfliktvermeidungsstrategie und verschiebe die temporäre Variable $T = CONTENTS(k)$ von Register k nach Register R. Alles andere geschieht wie bei der Konfliktlösungsstrategie.

Der mit dieser Strategie erzeugte Code vereinigt die Vorteile beider einzelnen Strategien:

Beispiel: `p(X, Y, f(Z)) ← q(a, b, Z, g(X, Y))`

Mit der letzten gemischten Strategie wird folgender Code erzeugt:

```
get_structure f/1, A3
unify_variable A3
get_variable X5, A1      % Loese Konflikt
put_constant a, A1
get_variable X6, A2      % Loese Konflikt
put_constant b, A2
put_structure g/2, A4
unify_value X5
unify_value X6
execute q/4
```

Vorteil der gemischten Strategie: Zur Konfliktlösung wird eine temporäre Variable, die als Kopfargument vorkommt, nur einmal in ein neues Register verschoben.

4.4.5 Verbesserte globale Registerallokation

Die bisherigen Verbesserungen der Registerallokation stützen sich nur auf eine lokale Analyse einer Klausel ab. Weitergehende Optimierungen kann man durch globale Analysen erreichen.

Beispiel: Gegeben sei die Klausel

```
p(X, Y, Z) ←  append(X, Y, W),  q(W, Z)
```

Nach dem bisherigen Schema sind `W` und `Z` permanente Variablen, wobei `Z` bei Anwendung dieser Klausel schon in einem Register steht. Statt nun `Z` als permanente Variable auf den Stack zu verschieben, könnte man `Z` auch in einem Register halten, da man weiß, daß ein Aufruf von `append` nie mehr als die ersten 4 Register verändert. Daher könnte `Z` im Register 5 während des `append`-Aufrufs gespeichert werden, d.h. der folgende Code ist eine korrekte Übersetzung dieser Klausel:

```
allocate
get_variable X5, A3   % Lade Z in Register 5
put_variable Y1, A3
call append/3, 1
```

```
put_unsafe_value Y1, A1
put_value X5, A2
deallocate
execute q/2
```

Probleme bei der automatischen Erzeugung eines solchen verbesserten Codes:

1. Welche permanenten Variablen sollen statt auf dem Stack in Registern gespeichert werden? Kandidaten hierfür sind solche Variablen, deren erstes Vorkommen im Kopf oder in einer Struktur liegt.
2. In welcher Reihenfolge soll der Code für alle Prädikate erzeugt werden? Vorschlag: Sortiere die Prädikate nach ihren gegenseitigen (Aufrufs-)Abhängigkeiten und erzeuge zuerst den Code für die unabhängigen Prädikate, dann für die nur davon abhängigen Prädikate etc.
3. Backtracking: Im obigen Beispiel darf `X5` auch durch nachfolgende Prädikate nicht verändert werden, falls der Code durch Backtracking noch einmal ausgeführt werden könnte.

Weitere Details findet man in [Deb86].

4.4.6 Umsortierung der Unifikationsreihenfolge

Die in [Deb86] vorgeschlagenen Optimierungen beziehen sich immer auf eine feste Reihenfolge beim Laden und Unifizieren von Argumenten: Die Argumente eines aufzurufenden Literals werden mittels **put**-Anweisungen immer von links nach rechts geladen, und genauso werden sie beim Aufruf mit einem Klauselkopf unifiziert. Da die Unifikation im Gegensatz zum Auswerten von Prozedurparametern in imperativen Sprachen keine Seiteneffekte hat und die Reihenfolge, in der die Argumente einer Struktur unifiziert werden, keinen Einfluß auf das Ergebnis hat, können die Argumente auch in einer anderen Reihenfolge geladen und unifiziert werden. Daher wird in [JDM88] eine Verbesserung vorgeschlagen, die auf einer anderen Reihenfolge basiert. Wir skizzieren hier nur anhand von Beispielen die grundsätzlichen Ideen. Die genauen Algorithmen hierzu kann man in der Originalarbeit nachlesen.

Betrachten wir zunächst Blöcke, die nicht als erste im Rumpf vorkommen, d.h. bei denen sich noch keine temporären Variablen in Argumentregistern befinden. Sei z.B.

```
p(X, f(X))
```

das letzte Literal in einem solchen Block. Der bisher erzeugte Code lautet (falls `X` hier zum ersten Mal auftritt)

```
put_variable X1, A1     % wie bisher gilt: Ai = Xi
put_structure f/1, A2
unify_value A1
```

Wenn jedoch der Code für das 2. Argument vor dem Code für das 1. Argument erzeugt wird, kann man folgenden verbesserten Code erzeugen:

```
put_structure f/1, A2
unify_variable A1
```

Die Strategie zur Codeerzeugung für das letzte Literal eines Blocks lautet (kombiniert mit einer geschickten Registerzuteilung):

1. Erzeuge zunächst den Code für die Argumente, die zusammengesetzte Terme sind und temporäre Variablen enthalten.
2. Erzeuge dann den Code für die Argumente, die temporäre Variablen sind.
3. Erzeuge zuletzt den Code für alle übrigen Argumente.

Einen Algorithmus, basierend auf dieser Strategie, ist in [JDM88] angegeben.

Schwieriger ist die Codeerzeugung für den ersten Block, da es hier aufgrund schon mit temporären Variablen belegter Register zu Konflikten kommen kann. Auch hier wird wieder die Unfikationsreihenfolge verändert:

1. Codeerzeugung für den Klauselkopf:
   a) Erzeuge zunächst den Code für die konstanten Argumente und die Argumente, die permanente Variablen sind.
   b) Erzeuge dann den Code für die Argumente, die temporäre Variablen sind.
   c) Erzeuge abschließend den Code für die übrigen Argumente (Strukturen).
2. Codeerzeugung für den ersten Block ohne das letzte Literal.
3. Codeerzeugung für das letzte Literal im ersten Block. Dies geschieht in der gleichen Reihenfolge wie bei anderen Blöcken (erst zusammengesetzte Terme, dann temporäre Variablen und abschließend Konstanten), jedoch mit einer geschickteren Konfliktlösungsstrategie.

Beispiel: Der erzeugte Code für die Klausel

```
p(X, Y, a) ←  q(X, b, f(Y))
```

lautet nach dieser Methode:

```
get_constant a, A3
put_structure f/1, A3
unify_value A2
put_constant b, A2
execute q/3
```

Dieser Code ist optimal. Zum Vergleich: Der Standardcode ohne Optimierungen (Kapitel 4.2) enthält drei zusätzliche `get`/`put`-Instruktionen.

Anhand der Klausel

```
p(U, V, W, X) ←   q(a, U, V, W, X)
```

stellen wir die beiden Arten der Codeerzeugung gegenüber:

Codeerzeugung nach Debray	Codeerzeugung mit veränderter Reihenfolge
`get_variable X6, A1`	`get_variable A5, A4`
`put_constant a, A1`	`get_variable A4, A3`
`get_variable X7, A2`	`get_variable A3, A2`
`put_value X6, A2`	`get_variable A2, A1`
`get_variable X6, A3`	`put_constant a, A1`
`put_value X7, A3`	`execute q/5`
`get_variable A5, A4`	
`put_value X6, A4`	
`execute q/5`	

4.5 Speicherbereinigung

Bei der Abarbeitung eines Prolog-Programms entstehen sehr viele neue Datenstrukturen, die zu unterschiedlichen Zeitpunkten wieder gelöscht werden:

1. Auf dem **lokalen Stack** werden Backtrackpunkte und Umgebungen angelegt. Backtrackpunkte werden gelöscht, sobald sie nicht mehr benötigt werden. Umgebungen werden verkleinert bzw. gelöscht, sobald alle alternativen Beweise der zugehörigen Klausel abgearbeitet sind. Jedoch können Umgebungen durch später angelegte Backtrackpunkte „eingefroren“ werden, so daß sie eventuell unnötigerweise auf dem Stack stehen bleiben. Dies kann durch Einfügen von Cuts verhindert werden. Mit Hilfe von Cuts und des automatischen „environment trimming“ und „last goal optimization“ wird der Speicherplatz im lokalen Stack gut verwaltet.

2. Auf dem **Heap** werden neue Strukturen und ungebundene Variablen gespeichert. Es kann aber häufig passieren, daß Datenstrukturen auf dem Heap vom Programm aus nicht mehr zugreifbar sind (z.B. wenn eine Datenstruktur zunächst über eine permanente Variable referenziert wurde, die aber später aufgrund des „environment trimming“ gelöscht wurde). Da der Heap standardmäßig nur beim Backtracking verkleinert wird, ist eine Speicherbereinigung auf dem Heap besonders lohnenswert.

3. Im **Codebereich** werden die übersetzten Klauseln gespeichert. Wenn bei der Programmabarbeitung viele **assert**- und **retract**-Literale ausgeführt werden, kann auch hier eine Speicherbereinigung notwendig sein.

Die meisten überflüssigen Daten fallen somit in der Regel auf dem Heap an. Daher sollte jede ernsthaft einsetzbare Prolog-Implementierung über eine automatische Speicherbereinigung auf dem Heap verfügen. Hierfür gibt es in der Literatur schon diverse Vorschläge, auf die wir im Rahmen dieser Vorlesung nicht mehr detailliert eingehen können. Daher seien an dieser Stelle nur einige Literaturstellen hierzu erwähnt: [ACHS88] [BM86] [Bru84] [Klu88] und [PBW85].

4.6 Benchmarks

Wichtigstes Ziel einer effizienten Prolog-Implementierung ist eine hohe Geschwindigkeit bei der Ausführung der Programme. Daneben sollte die Ausführung nicht unnötig viel Speicher benötigen und bei der Programmentwicklung ist auch die Geschwindigkeit des Compilers selbst relevant.

Zur Messung der Ausführungsgeschwindigkeit von logischen Programmen wird i.a. ein Maß benutzt, das aber nicht unumstritten ist: **LIPS** („logical inferences per second"), d.h. Anzahl der Inferenzschritte pro Sekunde. Dabei ist ein Inferenzschritt definiert als eine erfolgreiche Unifikation mit einem Klauselkopf (Klauselanwendung ohne Abarbeitung des Rumpfes).

Diese Maßeinheit ist deswegen umstritten, weil sie vom abzuarbeitenden Programm sehr stark abhängig ist: So erreicht ein Programm, in dem es zu jedem Prädikat höchstens zwei Klauseln gibt, eine sehr hohe LIPS-Zahl, während ein Programm, bei dem ein Prädikat, das durch mehrere tausend Klauseln definiert ist (wobei aber in der Regel nur eine paßt), häufig aufgerufen wird, nur eine extrem niedrige LIPS-Zahl erreicht.

Trotzdem wird häufig als Standardbenchmark das Programm „naive reverse" benutzt:

```
rev([], []) ←
rev([E|R], L) ← rev(R, UR), append(UR, [E], L)

append([], L, L) ←
append([E|R], L, [E|RL]) ← append(R, L, RL)
```

Die Umkehrung einer Liste mit N Elementen benötigt somit $\frac{(N+1)(N+2)}{2}$ Inferenzschritte. Der Benchmark wird gestartet, indem `rev` mit einer 30-elementigen Liste als erstem Argument und einer Variablen als zweitem Argument aufgerufen wird. Dies wird K-mal wiederholt (K ist groß genug, damit die Zeitmeßfehler klein bleiben) und dafür die Gesamtzeit T (in Sekunden) gemessen. Die LIPS-Zahl L lautet dann:

$$L = \frac{496 * K}{T}$$

Mit dieser Meßmethode erreichen heutige Prolog-Compiler Werte von mehr als 500 KLIPS (= 500.000 LIPS) auf herkömmlichen Rechnern. Allerdings sagt dies nur bedingt etwas über die Geschwindigkeit bei realistischen Anwendungen aus, denn:

- „Naive reverse" arbeitet vollkommen deterministisch und ist daher „Prolog-untypisch".
- Viele Compiler sind auf diesen Benchmark zugeschnitten und arbeiten ihn daher besonders schnell ab.
- Der WAM-Code für dieses Programm ist relativ klein. Bei der Ausführung kommt es nicht zu Seitenfehlern (Paging).

Sinnvoller sind daher Benchmarks, die verschiedene Aspekte einer Implementierung untersuchen, wie z.B. Speicherverbrauch durch Umgebungen, Anlegen von Backtrackpunkten, Unifikation etc. Ein Ansatz, der verschiedene Optimierungen in der WAM untersucht, befindet sich in [TD87].

Kapitel 5

Implementierung nicht-logischer Konstrukte

Die WAM, so wie sie im letzten Kapitel vorgestellt wurde, dient nur zur effizienten Ausführung rein logischer Hornklauselprogramme. Prolog-Programme enthalten in der Regel eine Vielzahl nicht-logischer Konstrukte (Cut, Arithmetik, Datenbankoperationen, vgl. [Han87]). Die Implementierungsmöglichkeiten dieser Konstrukte im Rahmen der WAM sollen in diesem Kapitel angedeutet werden.

5.1 „Cut“

Ein „Cut“ in einer Klausel bedeutet operational, daß an dieser Stelle die Möglichkeit des Backtracking für das gerade im Beweis befindliche Literal (das „Vaterziel“) verhindert wird. Bei unserem ersten deterministischen Interpreter (Kapitel 2) kann dies einfach dadurch implementiert werden, daß der Backtrackpunkt des Vatergoals und alle darüberliegenden Backtrackpunkte gelöscht werden. Dies liegt daran, daß es zu jedem zu beweisenden Literal einen Backtrackpunkt gibt.

In der WAM kann es jedoch zu einem Literal 0, 1 oder auch 2 Backtrackpunkte geben (vgl. Abschnitt 4.3.1). Daher ist die Implementierung des „Cut“ hier nicht so einfach. Wie kann man nun feststellen, welche Backtrackpunkte bei Bearbeitung eines „Cut“ gelöscht werden sollen? Eine einfache Möglichkeit ist in [BBCT86] angegeben: Man merkt sich bei Aufruf eines Literals den aktuellen Backtrackpunkt in einem speziellen Register (`CUT`). Bei Auftreten eines „Cut“ in einer Klausel muß man nur das Register `B` auf den Wert von `CUT` setzen, wodurch alle Backtrackpunkte, die nach Aufruf des Literals angelegt worden sind, wirkungslos werden. Zu beachten ist dabei, daß der Wert von `CUT` bei komplexeren Klauseln wie eine permanente Variable in der aktuellen Umgebung gespeichert werden muß.

Zur Realisierung dieser Idee muß die bisher vorgestellte WAM wie folgt modifiziert werden:

- Es wird ein neues WAM-Register `CUT` eingeführt, das immer auf einen Backtrackpunkt im Stack verweist.
- Die WAM-Instruktionen „`call`“ und „`execute`“ setzen zusätzlich immer `CUT` auf den aktuellen Wert von `B`.

- Jeder Backtrackpunkt wird um die Komponente `BCUT` erweitert. Bei Anlegen eines Backtrackpunkts wird der Wert von `CUT` in `BCUT` gespeichert und bei „`fail`“ wird `BCUT` in `CUT` zurückgespeichert.

Daneben werden folgende zusätzliche Instruktionen eingeführt:

Instruktion	**Bedeutung**
set_B_from_CUT	B := CUT
set_B_from Yn	B := Yn
save_CUT_in Yn	Yn := CUT

Die folgenden Beispiele zeigen die Benutzung dieser neuen Befehle:

Klausel: `p` ← `!`
Code:

```
set_B_from_CUT
proceed
```

Klausel: `p` ← `q1, !, q2`
Code:

```
allocate          % eine permanente Variable für CUT
save_CUT_in Y1
call q1/0, 1
set_B_from Y1
deallocate
execute q2/0
```

Klausel: `p` ← `q1, q2, !`
Code:

```
allocate
save_CUT_in Y1
call q1/0, 1
call q2/0, 1
set_B_from Y1
deallocate
proceed
```

Dieses Schema kann natürlich noch optimiert werden (z.B. muß bei `call` bzw. `execute` das Register `CUT` nur geladen werden, wenn in mindestens einer Klausel des aufgerufenen Prädikats ein „Cut“ vorkommt).

Die Verwendung von „Cut“ in Disjunktionen kann mit den gleichen Mechanismen realisiert werden, nur muß man sich hierbei erst einmal klar werden, was für eine Wirkung ein solches „Cut“ haben soll. Dies ist vielen Implementierern von Prolog eben nicht klar, wie in [Mos86] gezeigt wird: Chris Moss hat 24 Prolog-Implementierungen untersucht und dabei für ein identisches Programm mit „Cut“s 11 unterschiedliche Verhaltensweisen der Implementierung erhalten!

5.2 Arithmetik

Hierbei geht es im wesentlichen um die Implementierung des Prädikats `is/2` (wir betrachten hier nur Integer-Arithmetik). Zum Beweis eines `is`-Literals muß das 2. Argument (das ein arithmetischer Ausdruck sein muß) ausgerechnet und anschließend mit dem ersten unifiziert werden. Zur Implementierung wird eine Routine `eval` benötigt, die einen arithmetischen Ausdruck über den Aufbau des Terms auswertet. Die Zwischenergebnisse der rekursiven Prozedur `eval` können z.B. auf dem Hilfsstack PDL abgelegt werden.

Sei also

```
eval Xi, An
```

eine neue WAM-Instruktion, die den in `An` stehenden arithmetischen Ausdruck auswertet (dabei können Laufzeitfehler auftreten!) und das Ergebnis in `Xi` schreibt. Dann kann das Prädikat `is/2` wie folgt implementiert werden:

```
is/2: eval X3, A2
      get_value X3, A1
      proceed
```

Als weiteres Beispiel geben wir die Implementierung des arithmetischen Vergleichsprädikats `=:=/2` an:

```
=:=/2: eval A1, A1
       eval A2, A2
       get_value A1, A2
       proceed
```

In Prolog-Programmen kommen relativ häufig Literale der Form `X is Y+1` zum Inkrementieren von Werten vor. Bei der üblichen Implementierung würde erst die Struktur `+(Y,1)` auf dem Heap erzeugt werden und anschließend wird diese mit `eval` ausgewertet. Wenn man jedoch eine neue WAM-Instruktion

```
inc Vn, Ai
```

einführt, die die in `Ai` stehende Zahl um 1 erhöht (falls in `Ai` eine Zahl steht, sonst Laufzeitfehler) und das Ergebnis in `Vn` speichert, dann kann der Compiler ein solches `is`-Literal direkt in eine Folge von WAM-Instruktionen übersetzen:

Klausel: `inc(X, Y)` ← `Y is X+1`

Code:

```
inc/2: inc A1, A1
       get_value A1, A2
       proceed
```

Im Vergleich dazu der Code bei der herkömmlichen Implementierung:

```
inc/2: get_variable X3, A1
       put_value A2, A1
       put_structure +/2, A2
       unify_local_value X3
       unify_constant 1
       execute is/2
```

Diese Optimierung ist jedoch nur korrekt, wenn bei Aufruf von `inc` das 1. Argument immer eine Zahl (und kein arithmetischer Ausdruck) ist. Dies kann man eventuell durch statische Analysemethoden schon zur Übersetzungszeit feststellen (→ Kapitel 6).

Eine weitere Alternative ist die Übersetzung arithmetischer Ausdrücke in Code für eine Stackmaschine, d.h. die Argumente einer arithmetischen Operation werden immer auf einem arithmetischen Stack gespeichert und bestimmte Operationen (ADD, MULT) manipulieren die obersten Stackelemente. Weitere Details hierzu findet man in [Clo85a].

5.3 Metalogische Prädikate

Die Prädikate `var/1`, `nonvar/1`, `atom/1`, `integer/1` und `atomic/1` können einfach dadurch implementiert werden, daß zunächst das Etikett des Argumentterms überprüft wird und anschließend in Abhängigkeit dieses Wertes `proceed` bzw. `fail` ausgeführt wird.

Eine interessante Alternative ist in [Car87] angegeben: Der Compiler kann das Wissen über diese metalogischen Prädikate benutzen, um diese Prädikate in bestimmten Fällen direkt in das Indizierungsschema zu übersetzen. Wenn z.B. ein Prädikat durch die beiden Klauseln

```
p(X, Y) ←   var(X), q(X, Y)
p(X, Y) ←   nonvar(X), r(X, Y)
```

definiert ist, dann kann es in eine einzige Anweisung übersetzt werden (wie immer setzen wir voraus, daß der Code für ein Prädikat `p` der Stelligkeit `n` bei der Marke `p/n` beginnt):

```
p/2: switch_on_term q/2, r/2, r/2, r/2
```

Diese Optimierung ist natürlich nur möglich, wenn die mit `var/1` überprüfte Variable nur einmal im Klauselkopf vorkommt. Im nächsten Beispiel kann die Optimierung nicht angewendet werden:

```
q(X,X) ← var(X), v(X)
q(X,X) ← nonvar(X), w(X)
Anfrage: q(Z,1)
```

Wenn allerdings die `switch_on_term`-Anweisung so erweitert wird, daß sie nicht nur auf das Register `A1`, sondern auf jedes Register anwendbar ist (vgl. Abschnitt 4.4.2), dann kann jedes dieser metalogischen Prädikate direkt durch WAM-Instruktionen auscompiliert werden. Z.B. können dann die Klauseln

```
r(X, Y, Y) ← var(Y), s(X,Y)
r(X, Y, Y) ← nonvar(Y), t(X,Y)
```

wie folgt übersetzt werden:

```
r/2: get_value A3, A2
     switch_on_term A3, s/2, t/2, t/2, t/2
```

5.4 Datenbankmanipulationen

Die Prädikate zur Manipulation der Datenbank (`assert`, `retract`) sind nicht so einfach zu implementieren wie andere vordefinierte Prädikate, denn sie bedeuten, daß das Programm sich selbst während der Ausführung verändert. Diese im Vergleich zu herkömmlichen Programmiersprachen neue Möglichkeit verlangt auch völlig neue Implementierungstechniken. Einige Ideen hierzu werden in diesem Abschnitt vorgestellt.

Durch Hinzufügen (`assert`) oder Löschen (`retract`) von Klauseln wird das gesamte Prolog-Programm verändert. Daher bietet sich als einfachste Möglichkeit an, nach jeder Datenbankänderung das Programm neu zu übersetzen. Vorteil hiervon: Es wird zu jedem Zeitpunkt immer der optimale Code abgearbeitet. Nachteil: Da die Übersetzung mit allen Codeoptimierungen einige Zeit in Anspruch nimmt, ist diese Methode extrem ineffizient. Aus diesem Grund muß man auf eine komplette Neuübersetzung verzichten und stattdessen nach effizienteren Kompromissen suchen.

Eine einfache Möglichkeit ist, Klauseländerungen nur bei bestimmten Prädikaten zuzulassen. Diese Prädikate müssen vom Benutzer angegeben werden (z.B. durch „`dynamic`“-Deklarationen) und sie werden auch nicht übersetzt, sondern durch einen Interpreter abgearbeitet.

Vorteile:

- Einfache Technik (Implementierung)
- Schnelle Ausführung von `assert`/`retract`

Nachteile:

- Zusätzlicher Codebereich für nicht-übersetzte Klauseln
- Langsame Ausführung von `dynamic`-Prädikaten
- Zusätzliche Implementierung eines Interpreters (Konsistenz von Interpreter und Compiler!)
- Restriktiv: Man muß vorher alle Prädikate deklarieren, die evtl. verändert werden sollen

Eine Verbesserung, die einige dieser Nachteile beseitigt, ist in [Pro87] angegeben: Zu Beginn werden alle Prädikate compiliert und auch der Quellcode aller Klauseln gespeichert. Wenn ein Prädikat mit `assert` oder `retract` verändert wird, dann wird diese Änderung nur im Quellcode durchgeführt und gleichzeitig der übersetzte Code als ungültig markiert. Wenn nun dieses Prädikat ausgeführt wird, dann wird nicht mehr der WAM-Code, sondern der Quellcode mit einem Interpreter ausgeführt. Nach einigen Quellcode-Änderungen kann dieser z.B. wieder compiliert werden, wodurch dann später der WAM-Code ausgeführt wird.

Vorteile:

- Keine Restriktionen bei `assert/retract` (gleiches Verhalten wie bei Interpreter-Systemen)
- Keine grundsätzlich langsame Ausführung von Prädikaten, die mit `assert/retract` veränderbar sind

Weitere Effizienzsteigerungen kann man nur erreichen, indem tatsächlich der WAM-Code bei `assert` und `retract` verändert wird. Hierdurch spart man sich den Speicherbereich für den Quellcode und die Implementierung eines Interpreters.

Die einfachste Möglichkeit hierfür ist, alle Klauseln eines veränderbaren Prädikats ohne Indizierungsschema nur mit einer (`re`)`try_me_else`-Kette zu verbinden: Seien $K_1, \ldots, K_m$ alle Klauseln für ein veränderbares Prädikat `p/n`. Dann wird für `p/n` der Code nach folgendem Schema erzeugt:

```
p/n: goto L1          % goto ist neuer WAM-Befehl
L1:  try_me_else L2,n
     < Code für K_1 >
L2:  retry_me_else L3
     < Code für K_2 >
L3:  retry_me_else L4
     < Code für K_3 >
...
Lm:  trust_me_else fail
     < Code für K_m >
```

Dann kann `assert/retract` wie folgt implementiert werden:

1. `asserta(K)`: Sei `L` die Adresse des nächsten freien Speichers im Codebereich. Dann verändere den obigen Code wie folgt an zwei Stellen:

   ```
   p/n: goto L

   L1:  retry_me_else L2
   ```

   Füge folgenden Code hinzu:

   ```
   L: try_me_else L1,n
      < Code für Klausel K>
   ```

2. `assertz(K)`: Sei `L` die Adresse des nächsten freien Speichers im Codebereich. Dann verändere den obigen Code wie folgt:

   ```
   Lm: retry_me_else L
   ```

   Füge folgenden Code hinzu:

   ```
   L: trust_me_else fail
      < Code für Klausel K>
   ```

3. `retract(K)`: Sei Kj die zu löschende Klausel. Dann ändere den Code wie folgt:

   Falls $j = 1$:

   ```
   p/n:  goto L2

   L2:   try_me_else L3,n
   ```

   Falls $j = m$:

   ```
   Lm-1: trust_me_else fail
   ```

   Falls $j = 2$:

   ```
   L1:   try_me_else L3,n
   ```

   sonst:

   ```
   Lj-1: retry_me_else Lj+1
   ```

Vorteile dieses Schemas:

- Compilation von höchstens einer Klausel (damit dies schnell geht, kann man z.B. die Optimierungsphase im Compiler weglassen)
- Einfach zu realisieren, schnelle Ausführung von `p`-Literalen auch nach Veränderung der Klauseln.

Nachteil:

- Merkwürdige Semantik: Klauselveränderungen werden zu unterschiedlichen Zeitpunkten wirksam (`assertz` wird sofort wirksam bei Beweisen, die gerade die Klauseln $K_1, \ldots, K_{m-1}$ benutzen, aber nicht bei Verwendung der Klausel K_m).

Dieser semantische Aspekt wurde auch in [Mos86] untersucht. Da es keine einheitliche Festlegung der Wirkung von `assert/retract` gibt, implementiert es jeder anders. So erhielt Moss beim Testen von 12 verschiedenen Prolog-Systemen 9 verschiedene Antworten!

Für eine einheitliche Sichtweise wird auch in [LO87] plädiert. Sie geben folgende Klassifikation der Sichtweisen an:

- Bei der **logischen Sichtweise** wird ein Literal genau mit den Klauseln bewiesen, die zum Zeitpunkt des Beweisbeginns dieses Literals vorhanden sind. Änderungen in der Klauselfolge, die während des Beweises für dieses Literal vorgenommen werden, sind somit für den Beweis dieses Literals nicht sichtbar.
- Bei der **Sichtweise der sofortigen Änderung** ist jede Änderung an der Klauselfolge bei einem Backtracking-Schritt immer sichtbar.

- Die **Implementierungssichtweise** liegt irgendwo zwischen diesen beiden Extremen. Unsere obige Implementierung gehört zu dieser Kategorie.

Beispiel: Gegeben seien die folgenden Klauseln:

```
p ← assertz(p), fail
p ← fail

q ← fail
q ← assertz(q), fail
```

Bei der logischen Sichtweise erhält man auf die Anfrage „`?-p`“ wie auch auf die Anfrage „`?-q`“ die Antwort „no“.

Bei der Sichtweise der sofortigen Änderung erhält man auf diese beiden Anfragen die Antwort „yes“.

In C-Prolog [Per87] ist die Implementierungssichtweise realisiert: Man erhält bei „`?-p`“ die Antwort „yes“ und bei „`?-q`“ die Antwort „no“ (dieses Verhalten ergibt sich auch bei unserer obigen Implementierung).

Die Implementierungssichtweise ist aus Gründen der Portabilität abzulehnen. Die Sichtweise der sofortigen Änderung macht viele Optimierungen der WAM hinfällig und führt zu einer schlechten Speicherausnutzung: Während die WAM versucht, das Anlegen von Backtrackpunkten zu vermeiden, sind diese bei der Sichtweise der sofortigen Änderung immer notwendig. Backtrackpunkte könnten bei dieser Sichtweise erst gelöscht werden, wenn ein Beweisversuch für ein Literal komplett fehlschlägt (oder bei einem „Cut“). Existiert z.B. für das Prädikat `p` nur die Klausel

```
p ←  assertz(p), fail
```

dann würde die WAM für den Beweis des Literals `p` keinen Backtrackpunkt anlegen. Bei der Sichtweise der sofortigen Änderung muß aber auf jeden Fall ein Backtrackpunkt angelegt werden, sonst ist `p` nicht beweisbar. Somit ist aus Effizienzgründen diese Sichtweise ungeeignet.

Aus diesen Gründen ist die logische Sichtweise die einzige akzeptable. Wie kann diese implementiert werden? Einfache Möglichkeit: Immer wenn ein Literal bewiesen werden soll, dann wird eine neue Kopie aller zu diesen Zeitpunkten zur Verfügung stehenden Klauseln (des entsprechenden Prädikats) angelegt und der Beweis wird nur mit den kopierten Klauseln geführt. Dieses Schema haben wir bei unseren ersten deterministischen Interpretern benutzt, aber es ist natürlich viel zu ineffizient.

Eine bessere Lösung ist in [LO87] angegeben: Statt Klauseln bei einem Beweis wirklich zu kopieren, wird eine „*virtuelle Kopie*“ angelegt. Eine virtuelle Kopie einer Klausel besitzt als zusätzliche Information einen Zeitstempel, der angibt, ob die Klausel nun gültig ist oder nicht. Ein **Zeitstempel** ist ein Intervall $(Birth, Death]$, das angibt, von welcher Zeit bis zu welcher Zeit die Klausel existiert. Prinzipiell ist dabei folgendes zu implementieren:

1. Das Prolog-System ist mit einer „Uhr“ ausgestattet.

2. Wenn eine neue Klausel hinzugefügt wird, erhält sie den Zeitstempel $(AT, \infty]$, wobei AT die aktuelle Systemzeit ist (∞ ist eine beliebig weit entfernte Zeit in der Zukunft).

3. Wenn eine Klausel gelöscht wird, dann wird ihr Zeitstempel $(T1, T2]$ geändert zu $(T1, AT]$, wobei AT die aktuelle Systemzeit ist.

4. Beim Beweisbeginn für ein Literal wird die zu diesem Zeitpunkt aktuelle Uhrzeit gespeichert. Wenn zum Beweis dieses Literals eine Klausel mit dem Zeitstempel $(T1, T2]$ ausprobiert werden soll, so wird sie nur benutzt, falls die Uhrzeit des Literals in diesem Intervall liegt.

Man kann sich leicht überlegen, daß durch dieses Schema die logische Sichtweise realisiert wird.

Implementierung in der WAM:

- Führe neue WAM-Register `CC` („call clock“) und `GC` („global clock“) ein. `GC` entspricht der System-Uhr und kann ganzzahlige Werte annehmen. `GC` wird bei jedem `assert/retract` automatisch um eins erhöht. Das Register `CC` ist eine für jedes Literal lokale Größe und wird daher im Backtrackpunkt gespeichert und bei einem „`fail`“ wieder zurückgespeichert.

- Führe eine neue WAM-Instruktion

  ```
  dynamic_else Birth, Death, Next
  ```

  ein. Sie ersetzt bei veränderbaren Prädikaten die Instruktionen `try_me_else`, `retry_me_else` und `trust_me_else`, wobei (`Birth`, `Death`] der Zeitstempel der Klausel ist, vor der diese Anweisung steht, und `Next` die Adresse der nächsten Klausel ist. Diese neue Instruktion arbeitet in zwei Modi: Im „first“-Modus, wenn dies die erste Klausel eines veränderbaren Prädikats ist, und im „next“-Modus, wenn dies eine weitere Klausel ist (d.h. der „next“-Modus wird beim Backtracking erreicht).

  Die `dynamic_else`-Anweisung wirkt wie folgt: Im „first“-Modus wird zunächst das Register `CC` auf den Wert von `GC` gesetzt (d.h. das Literal erhält seine Aufrufzeit). Dann wird geprüft, ob `CC` größer als `Birth` und kleiner oder gleich `Death` ist (in diesem Fall heißt die Klausel „lebend“). Falls nicht, wird dies bei der nächsten Klausel (adressiert über `Next`) geprüft usw. Wird keine Klausel gefunden, dann schlägt der Beweis fehl. Andernfalls wird, bevor diese Klausel angewendet wird, die nächste lebende Klausel für dieses Literal gesucht. Falls eine solche existiert, wird hierfür ein Backtrackpunkt (vgl. `try_me_else`) inklusive `CC` angelegt. Anschließend wird die Klausel ausgeführt.

  Im „next“-Modus (d.h. wenn eine vorhergehende `dynamic_else`-Anweisung einen Backtrackpunkt angelegt hat, durch den jetzt diese Anweisung angesprungen wird) wird zunächst die nächste lebende Klausel für dieses Literal gesucht. Falls eine solche existiert, wird ihre Adresse der Komponente `BP` im letzten Backtrackpunkt zugewiesen (`retry_me_else`). Falls keine nächste lebende Klausel existiert, dann wird der aktuelle Backtrackpunkt gelöscht (`trust_me_else`). Anschließend wird die Klausel ausgeführt.

Als Beispiel betrachten wir noch einmal die Klauseln

```
p ← assertz(p), fail
p ← fail

q ← fail
q ← assertz(q), fail
```

für die veränderbaren Prädikate `p` und `q`. Wenn diese Klauseln nacheinander eingelesen werden, ergibt sich z.B. der folgende Code:

```
p/0: dynamic_else 30, ∞, L1
     < Code für assertz(p)>
     fail
L1:  dynamic_else 31, ∞, fail
     fail

q/0: dynamic_else 32, ∞, L2
     fail
L2:  dynamic_else 33, ∞, fail
     < Code für assertz(q)>
     fail
```

Beim Beweis von `p` zur Zeit `GC` $= 50$ würde durch das `assertz(p)` das dritte Argument von `dynamic_else` bei `L1` von `fail` in `L3` geändert, wobei `L3` eine neue Codeadresse mit

```
L3: dynamic_else 50, ∞, fail
    proceed
```

ist. Diese Klausel wird beim Backtracking jedoch noch nicht gefunden, da das Literal `p` eine Aufrufzeit `CC` $= 50$ hat, und diese ist nicht größer als das Geburtsdatum dieser neuen Klausel.

Vorteile dieser Implementierung:

- kein unnötiger Speicherbedarf, da Backtrackpunkte nur bei Bedarf angelegt und frühzeitig gelöscht werden. Z.B. benötigt die Abarbeitung eines als „dynamisch“ deklarierten `append`-Prädikats nur die gleichen Backtrackpunkte wie früher angegeben.
- Schnelle Ausführung von `assert/retract`.
- Diese Technik kann auch mit einem Indizierungsschema verbunden werden (Details hierzu sind in [LO87] angegeben).

Ein Problem ergibt sich, falls die Systemuhr `GC` überläuft, weil laufend `assert/retract`-Operationen ausgeführt werden. In diesem Fall muß ein „Zeit-Kompaktifizierer“ alle Backtrackpunkte im lokalen Stack und alle dynamischen Klauseln durchgehen und ihre Zeitstempelwerte verkleinern.

5.5 Rekonstruktion des Quellcodes

Bei einigen vordefinierten Prädikaten (`retract`, `clause`, `listing`) und bei Programmierumgebungen (Debugger) ist es notwendig, auf den Quellcode der Klauseln zuzugreifen. Bei einem compilierenden Prolog-System gibt es folgende Möglichkeiten zur Lösung dieses Problems:

1. Neben dem übersetzten WAM-Code wird auch der Quellcode in einem separaten Bereich gespeichert. Normalerweise wird immer der WAM-Code ausgeführt, nur bestimmte Prädikate wie `retract`, `clause` und `listing` greifen auf den Quellcode zu bzw. beim Debugging wird der Quellcode mit einem Interpreter ausgeführt. Diese Lösung erfordert zusätzlichen Speicheraufwand, birgt Konsistenzprobleme zwischen Quellcode und WAM-Code und ist zudem langsam, da die Unifikation bei `retract` und `clause` nicht auscompiliert ist.

2. Für jede Klausel `K` wird ein Faktum `source(P,N,K,R)` compiliert [Clo85b], wobei `R` ein Verweis auf den übersetzten Code der Klausel `K` ist, und `P` bzw. `N` der Name bzw. Stelligkeit des Prädikats ist. Dann ist z.B. die Implementierung von `clause` im Ablauf sehr effizient, da hierzu die üblichen Mechanismen der WAM ausgenutzt werden, denn `clause` kann wie folgt implementiert werden:

   ```
   clause(H,B) ←   functor(H,P,N), source(P,N,(H ←   B), _)
   ```

   Diese Lösung führt aber auch zu einer Verdoppelung der Codegröße, zu einer Verdoppelung der Übersetzungszeit (es müssen die Klauseln `K` und `source(P,N,K,R)` compiliert werden) und birgt ähnliche Konsistenzprobleme wie Lösung 1.

3. Es wird nur der übersetzte Code gespeichert. Der Quellcode wird durch Decompilation (Rückübersetzung) rekonstruiert. Diese Lösung ist speichereffizient und vermeidet auch die Konsistenzprobleme zwischen Interpreter und Compiler. Einen effizienten Mechanismus zur Decompilation werden wir nun genauer vorstellen (dies lehnt sich an [Bue86] an).

Die Idee zur Decompilation ist folgende: *Der WAM-Code kann auch dazu benutzt werden, um den Quellcode zu rekonstruieren.* Z.B. gilt vor Ausführung einer `call`- oder `execute`-Instruktion, daß in den Argumentregistern die Argumentterme des entsprechenden Literals stehen, weil sie zuvor durch `put`-Instruktionen in die Register `Ai` geschrieben werden. Wenn man nun statt der Ausführung des `call` bzw. `execute` eine neue Struktur erzeugt, deren Funktor der Name des Literals und deren Argumente die Argumentregister sind, dann hat man dadurch den Code decompiliert. Wir werden diese Idee im folgenden konkretisieren, indem wir zeigen, wie man durch eine einfache Transformation des WAM-Codes einer Klausel einen neuen WAM-Code erhält, der die Klausel decompiliert. Anschließend werden wir Methoden besprechen, die ohne eine explizite Codetransformation auskommen.

Die Idee dieser Codetransformation wollen wir zunächst an einem einfachen Beispiel motivieren. Es sei die folgende Klausel gegeben:

```
p(a) ← q(b), r(c)
```

Diese Klausel wird normalerweise in den folgenden WAM-Code übersetzt:

```
allocate
get_constant a, A1
put_constant b, A1
call q/1, 0
put_constant c, A1
deallocate
execute r/1
```

Dieser Code wird jetzt modifiziert, so daß bei Ausführung des neuen Codes das Register `A1` mit dem Term, der dem Klauseltext entspricht (d.h. `:-(p(a), ’,’(q(b), r(c)))`), unifiziert wird. Dazu werden die `call`- und `execute`-Instruktionen durch neue WAM-Instruktionsfolgen ersetzt und einige Instruktionen am Klauselanfang eingefügt (die alten Codestücke sind eingerahmt):

```
allocate
get_structure ’:-’/2, A1        % Erzeuge Klauselstruktur
unify_variable X2
unify_variable X3
get_structure p/1, X2           % Erzeuge Kopfliteral p(a)
unify_variable A1
get_constant a, A1
put_constant b, A1
get_structure ’,’/2, X3         % Transformation von "call"
unify_variable X2
unify_variable X3
get_structure q/1, X2           % Erzeuge Literal q(b)
unify_local_value A1
put_constant c, A1
deallocate
get_structure r/1, X3           % Erzeuge Literal r(c)
unify_local_value A1
proceed
```

Im allgemeinen wird eine Klausel der Form

$$L_0 \leftarrow L_1, \ldots, L_m$$

in eine Codefolge der folgenden Form übersetzt (wir beachten zunächst die Spezialfälle für $m < 2$ nicht):

```
allocate
<get-Instruktionen für L_0 >
<put-Instruktionen für L_1 >
call p_1/n_1, e_1
<put-Instruktionen für L_2 >
call p_2/n_2, e_2
.
.
<put-Instruktionen für L_m >
```

```
deallocate
execute p_m/n_m
```

(Hierbei ist angenommen, daß p_i/n_i das Prädikat des Literals L_i ist).

Zur Decompilation wird im wesentlichen der Code für `call` und `execute` verändert: Statt die Prädikate aufzurufen, wird eine neue Struktur aufgebaut, die diesem Literal entspricht. Zusätzlich muß am Anfang der Klausel Code zur Erzeugung des Kopfliterals eingefügt werden. Der Klauselcode wird daher wie folgt modifiziert (die alten Codestücke sind eingerahmt):

```
allocate
get_structure ':-'/2, A1        % Erzeuge Klauselstruktur
unify_variable As
unify_variable At
get_structure p_0/n_0, As       % Erzeuge Kopfliteral
unify_variable A1
.
.
unify_variable An_0
<get-Instruktionen für L0>
<put-Instruktionen für L1>
get_structure ','/2, At         % Transformation von "call"
unify_variable As
unify_variable At
get_structure p_1/n_1, As       % Erzeuge Literal L_1
unify_local_value A1
.
.
unify_local_value An_1
<put-Instruktionen für L_2 >
.
.
<put-Instruktionen für L_m >
deallocate
get_structure p_m/n_m, At       % Erzeuge Literal L_m
unify_local_value A1
.
.
unify_local_value An_m
proceed
```

Anmerkungen:

- `As` und `At` sind neue, im alten Code nicht benutzte temporäre Variablen.
- Am Anfang der Klausel werden Instruktionen zur Erzeugung der Struktur

  `:-(`p_0`(_,...,_), _)`

eingefügt. Für jedes „`call`“ werden Instruktionen zur Erzeugung der Struktur

`','(`p_i`(_,...,_), _)`

eingefügt, wobei die Argumente von p_i mit den Argumentregistern, in denen die Prolog-Terme aus dem Quellcode stehen, unifiziert werden. Beim letzten „`execute`“ braucht die Struktur `','(_, _)` nicht erzeugt zu werden.

- Dieses Schema behandelt nur den Fall von Klauseln mit mindestens zwei Literalen im Rumpf, wobei jedes Literal mindestens ein Argument hat. Man kann es sehr einfach auf die Behandlung anderer Fälle erweitern: z.B. muß bei Erzeugung eines Literals L_i ohne Argumente ($n_i = 0$) statt „`get_structure`“ der Befehl `get_constant` p_i`,A`t benutzt werden.

- Der gesamte neue Code beschreibt ein Faktum, das den Quellcode als Argument enthält, d.h. wird dieser Code mit einer ungebundenen Variablen in `A1` aufgerufen, dann ist danach `A1` an einen Prolog-Term gebunden, der die Klausel

  $$L_0 \leftarrow L_1, \ldots, L_m$$

  repräsentiert. Somit kann man diesen Code als Übersetzung des Faktums

  `source(:-(`L_0`, (`L_1`, ..., `L_m`))) ←`

  auffassen. Somit ist klar, daß durch diese Transformation der Quellcode wieder zur Verfügung steht, obwohl er nicht explizit gespeichert wird.

Diese Lösung der Transformation des WAM-Codes führt zu einer Verdoppelung der Codegröße, was wir aber unbedingt vermeiden wollten. Wir brauchen aber den Code nicht explizit zu transformieren, wenn wir die WAM entsprechend modifizieren. Die o.a. Transformation ist ja nur eine Änderung der call- und execute-Instruktionen und des Klauselbeginns. Dies kann aber auch zur Laufzeit durchgeführt werden, wobei es dazu zwei Möglichkeiten gibt:

1. Definiere einen neuen „decompile“-Modus für die WAM durch Hinzufügen eines neuen (booleschen) Registers `DECOMPILE`. Wenn `DECOMPILE` = false ist, dann arbeitet die WAM wie bisher. Wenn jedoch `DECOMPILE` = true ist, dann werden die `call`- und `execute`-Instruktionen und der Klauselbeginn im Sinne der o.a. Transformation abgearbeitet (evtl. ist es günstig, den zusätzlichen Code für den Klauselbeginn explizit zu generieren, ihn aber nur bei `DECOMPILE` = true abzuarbeiten).

2. Setze zur Decompilation einer Klausel einen Breakpoint auf die erste `call`- oder `execute`-Instruktion. Bei Erreichen des Breakpoint wird die `call`- oder `execute`-Instruktion entsprechend der o.a. Transformation ausgeführt und ein neuer Breakpoint auf die nächste `call`- oder `execute`-Instruktion gesetzt (falls vorhanden).

Die erste Methode ist besser geeignet für Hardware-Implementierungen der WAM, weil dann das `DECOMPILE`-Register durch ein Modus-Bit realisiert werden kann. Da das ständige Abfragen des `DECOMPILE`-Registers ineffizient ist, ist bei einer Emulation der WAM die zweite Methode zu empfehlen. Dies ist

z.B. im portablen Prolog-System, das an der Syracuse University implementiert wurde [BBCT86], gemacht worden.

Problematisch wird diese Decompilierungstechnik bei optimiertem Code, den der Prolog-Compiler evtl. erzeugt, da dann nicht unbedingt für jedes Literal im Quellcode eine `call`- oder `execute`-Anweisung im WAM-Code stehen muß (vgl. die Definition von „Block“ in Abschnitt 4.4.4). Z.B. erzeugen viele Compiler für das Literal $t_1 = t_2$ im Rumpf einer Klausel den folgenden (schematischen) Code:

```
<put t1>, A1
<put t2>, A2
get_value A1, A2
```

Ebenso können sich `var`- und `nonvar`-Literale nicht explizit im Code widerspiegeln, sondern in das Indizierungsschema integriert werden (vgl. Abschnitt 5.3).

Zur Decompilation von solchen optimierten Klauseln muß der Decompiler die Optimierungen, die der Compiler durchführt, kennen, um dann z.B. das obige „`get_value`“ in einen Aufruf von `=/2` zu übersetzen. Eine andere Möglichkeit ist, bei Codeoptimierungen zusätzlich auch den nichtoptimierten Code zu speichern: Vor den optimierten Codestücken befindet sich dann ein bedingter Sprung, der nur bei der Decompilation ausgeführt wird und zum nichtoptimierten Code führt. Der Code für das Literal t_1 `=` t_2 könnte dann so aussehen:

```
        jump_if_decompile L2
        <put t1>, A1
        <put t2>, A2
        get_value A1, A2
L1:     ...
...
L2:     <put t1>, A1
        <put t2>, A2
        call =/2, 3
        jump L1
```

Diese Technik erlaubt es, einen einfachen Decompiler mit einem sehr stark optimierenden Compiler zu verbinden und die Vorteile beider Techniken zu vereinen. Dies wurde z.B. im KA-Prolog-System implementiert [LBD+87] [Nei86].

Kapitel 6

Optimierung durch statische Analyse

In Abschnitt 4.4 haben wir einige Optimierungstechniken kennengelernt. Dies waren im wesentlichen **lokale Optimierungen**, da sie nur durch Analyse eines lokalen Kontextes (einer Klausel) durchgeführt werden konnten. Dagegen benötigen **globale Optimierungen** [Mel85] Informationen über das gesamte Programm. Zu diesen Informationen gehören

- Modi von Prädikaten: Sind bestimmte Argumente bei Aufruf eines Prädikats immer nichtinstantiiert, instantiiert oder immer mit einem Grundterm instantiiert?
- Determiniertheit von Prädikaten: Werden bestimmte Prädikate vollkommen deterministisch, d.h. ohne Backtracking abgearbeitet?
- Variablen„sharing“: Ist bei Aufruf eines Prädikats garantiert, daß bestimmte Argumente (Variablen) an grundsätzlich verschiedene Terme gebunden sind?
- Argumenttypen: Von welchem Typ (z.B. Listen, Atome, Zahlen) sind die Argumente bei Aufruf eines Prädikats?

Diese Informationen kann man in der Regel nur durch eine globale Analyse des gesamten Programms ableiten, indem die Abhängigkeiten (welches Prädikat wird von wo wie aufgerufen?) im Programm analysiert werden. Im Prinzip muß man zur Gewinnung dieser Informationen *zur Compilezeit herausfinden, wie sich das Programm zur Laufzeit verhält.* Da dies prinzipiell unentscheidbar ist, hat man nach entscheidbaren Approximationen gesucht. Techniken hierzu werden in diesem Kapitel vorgestellt.

6.1 Verwendung von Laufzeitinformationen zur Übersetzung

Was kann man nun mit diesen Informationen anfangen? Eine Möglichkeit haben wir schon in Kapitel 3 kennengelernt: Programme können durch partielle Auswertung optimiert werden, indem Berechnungen, die normalerweise zur Laufzeit erfolgen, schon zur Compilezeit erledigt werden. Zur Anwendung von Transformationsregeln der partiellen Auswertung benötigt man teilweise Laufzeitinformationen. Z.B. kann man „`var(X)`“ durch „`true`“ ersetzen, wenn die Variable `X` immer ungebunden ist. Wesentlich effektiver kann man jedoch globale Informationen zu einer besseren Codeerzeugung bei der Compilation einsetzen [Mel85]. Wir geben hierzu einige Beispiele an.

1. **Modusinformationen:** Im DEC-10-Prolog-System kann der Benutzer zu jedem Prädikat angeben, welchen Modus die einzelnen Argumente beim Aufruf haben. Hierbei gibt es die folgenden drei Modi:
   '+': Das Argument ist immer ein Grundterm.
   '–': Das Argument ist immer eine ungebundene Variable.
   '?': Das Argument ist ein beliebiger Term.

   Diese Modusinformationen (oder auch noch speziellere) können durch abstrakte Interpretationstechniken auch automatisch abgeleitet werden [BJCD87] [BJ88] [DW88] [Som87] [Bru91]. Die Codeerzeugung kann durch Modusinformationen wie folgt verbessert werden:

   - Wenn das erste Argument (dies kann auf andere Argumente verallgemeinert werden) den Modus '+' hat, dann können die passenden Klauseln allein über das Indizierungsschema gefunden werden, d.h. die (`re`)`try_me_else`- und `trust_me_else`-Instruktionen, die die Klauseln sequentiell verknüpfen, brauchen nicht generiert zu werden.
   - Wenn das erste Argument den Modus '–' hat, dann ist das Indizierungsschema immer wirkungslos. Entweder man indiziert über ein anderes Argument, oder man läßt es ganz weg.
   - Wenn der genaue Modus ('+' oder '–') eines Argumentes bekannt ist, dann ist auch schon zur Compilezeit bekannt, ob die `unify`-Instruktionen für dieses Argument im Lese- oder Schreibmodus arbeiten. In diesem Fall kann der Compiler den spezialisierten Code erzeugen (vgl. Abschnitt 4.4.3).
   - In analoger Weise kann der Compiler auch spezielle `get`-Anweisungen erzeugen, wenn der Modus der entsprechenden Argumente bekannt ist. Z.B. kann die Anweisung `get_constant C,Ai` zu einer Abfrage (`C==Ai?`) bzw. einer Zuweisung (`Ai:=C`) reduziert werden, wenn der Modus des i. Argumentes bekannt ist ('+' bzw. '–').

2. **Determiniertheit von Prädikaten:** Wenn ein Prädikat deterministisch abgearbeitet wird, dann braucht hierfür kein Backtrackpunkt angelegt zu werden. In der Regel erledigt dies schon der Indizierungsmechanismus der WAM. Eventuell können aber auch deterministische (Teil-) Programme in funktionale Programme übersetzt werden [Red84], die ihrerseits effizienter implementierbar sind.

3. **Funktionale Berechnungen:** Ein Prädikataufruf besitzt nur funktionale Berechnungen, wenn die Ergebnismenge einelementig ist. Z.B. ist das Prädikat `p` definiert durch

   ```
   p(a) ←
   p(X) ← p(X)
   ```

   nicht deterministisch, aber jedes `p`-Literal besitzt nur funktionale Berechnungen, da die Lösungsmenge {`p(a)`} ist. Funktionale Berechnungen brauchen nur einmal ausgeführt zu werden (wenn sie keine Seiteneffekte haben). Sie können daher durch Einfügen von Cuts optimiert werden: Ist in der Klausel

   ```
   p(···) ←  q(···), r(···)
   ```

das Literal `q(···)` eine funktionale Berechnung, dann kann der folgende Code erzeugt werden, der ein Backtracking für `q(···)` verhindert (vgl. Implementierung von „Cut“ in Abschnitt 5.1):

```
< Code für Klauselkopf p(···)>
save_B_in Y1                  % Neue Instruktion Y1 := B
< Code für q(···)>
set_B_from Y1
< Code für r(···)>
```

Weitere Einzelheiten findet man in [DW89].

4. **Variablen„sharing“:** Wenn zur Compilezeit bekannt ist, daß in zu unifizierenden Termen verschiedene Variablen nicht an Terme mit gemeinsamen Variablen gebunden werden, dann kann die Unifikation beschleunigt werden, indem zur Laufzeit auf den Vorkommenstest („occur check“) verzichtet wird. Techniken zur Herleitung dieser Informationen findet man in [Pla84] und [Søn86].

5. **Typinformation:** Durch abstrakte Interpretation kann man teilweise Informationen darüber herleiten, von welchem Typ die Argumente eines Prädikataufrufs sind [BJ88]. Ein **Typ** ist dabei eine (evtl. unendliche) Menge von Termen. Eine einfache Typklassifikation ist z.B.

   type ground = <Menge aller Grundterme>
   type free = <Menge aller Variablen>
   type any = <Menge aller Terme>

   Aus dieser Typklassifizierung kann man Modusdeklarationen ableiten und entsprechende Optimierungen vornehmen (s.o.).

   Eine feinere Typklassifikation erhält man, wenn man Terme nach ihren Hauptfunktoren klassifiziert, wie z.B.

   type `list` = {`[]`} ∪ {•`(E,L)` | `E, L` beliebig}

   Hat man z.B. abgeleitet, daß der Aufruftyp von `append` immer `(list, list, free)` ist, dann kann die `switch_on_term`-Anweisung wie folgt vereinfacht werden:

```
<dereferenziere A1>
IF <Etikett von A1> = list
THEN <Führe 2. Klausel aus>
ELSE <Führe 1. Klausel aus>
FI
```

   Dabei kann die `try_me_else`/`trust_me_else`-Kette entfallen und der Code für die beiden Klauseln kann wie folgt verbessert werden:

   - In der 1. Klausel kann die Anweisung „`get_nil A1`“ entfallen.
   - In der 2. Klausel arbeiten die ersten beiden `unify`-Anweisungen im Lesemodus und die beiden letzten `unify`-Anweisungen im Schreibmodus (vgl. Abschnitt 4.4.3).

Durch Typinformationen können also das Indizierungsschema und die Unifikationsanweisungen für den Klauselkopf vereinfacht werden.

Diese Beispiele zeigen einige Möglichkeiten, wie man durch Informationen über den globalen Programmablauf den erzeugten Code verbessern kann. Weitere Möglichkeiten und ihre praktischen Auswertungen findet man in [MJMB89].

Wie kann man nun solche Eigenschaften aus dem gegebenen Programm ableiten? Zwei exakte Möglichkeiten wären:

1. Lasse das Programm (mit einer Anfrage) ablaufen und sammle dabei die gewünschte Information (Modi, Typen) auf. Der Programmablauf kann z.B. durch einen speziellen Interpreter erfolgen.

   Diese Möglichkeit ist nicht praktikabel, denn bei eventuellen Endlosschleifen würde auch diese Analysephase nicht enden. Dies bedeutet, daß der Compiler evtl. nicht terminiert, was natürlich nicht akzeptabel ist.

2. Beweise die gewünschten Programmeigenschaften durch eine formale Induktion über den Programmablauf.

   Beispiel: Gegeben sei das folgende Programm zur Umkehrung einer Liste:

   ```
   rev([], []) ←
   rev([E|R], L) ← rev(R, UR), append(UR, [E], L)
   append([], L, L) ←
   append([E|R], L, [E|RL]) ← append(R, L, RL)
   ?- rev([a1,...,an], L)
   ```

   Wir wollen zeigen, daß nach einem erfolgreichen Beweis dieser Anfrage die Variable `L` immer an eine Liste der Länge n gebunden ist. Dazu benötigen wir die folgende Aussage:

   Lemma: Nach einem erfolgreichen Beweis von `append([`a_1`,...,`a_n`], [`b_1`,...,`b_m`], L)` ist die Variable `L` an eine Liste der Länge $n + m$ gebunden.

   Beweis: Durch Induktion über die Länge n der ersten Liste:

   $n = 0$: Zum Literal `append([], [`b_1`,...,`b_m`], L)` paßt nur die 1. `append`-Klausel. Daher wird `L` an die $0 + m$-elementige Liste `[`b_1`,...,`b_m`]` gebunden.

   $n \Rightarrow n + 1$: Zum Literal `append([`a_1`,...,`a_{n+1}`], [`b_1`,...,`b_m`], L)` paßt nur die 2. `append`-Klausel. Daher wird im nächsten Schritt die Variable `L` an `[`a_1`|RL]` gebunden und das Literal `append([`a_2`,...,`a_{n+1}`], [`b_1`,...,`b_m`], RL)` versucht zu beweisen. Da das erste Argument eine n-elementige Liste ist, wird nach Induktionsvoraussetzung die Variable `RL` an eine $n + m$-elementige Liste gebunden. Somit wird insgesamt die Variable `L` an eine $(n + 1) + m$-elementige Liste gebunden. □

   Nun können wir unsere gewünschte Aussage beweisen:

   Satz: Nach einem erfolgreichen Beweis von `rev([`a_1`,...,`a_n`], L)` ist die Variable `L` an eine Liste der Länge n gebunden.

Beweis: Durch Induktion über die Länge n der Liste:

$n = 0$: Zum Literal `rev([], L)` paßt nur die 1. `rev`-Klausel. Daher wird `L` an die 0-elementige Liste `[]` gebunden.

$n \Rightarrow n+1$: Zum Literal `rev([`a_1`,...,`a_{n+1}`], L)` paßt nur die 2. `rev`-Klausel. Daher wird im nächsten Schritt das Ziel

```
?- rev([a2,...,an+1], UR), append(UR, [a1], L).
```

versucht zu beweisen. Da das erste Argument vom `rev`-Literal eine n-elementige Liste ist, wird nach Induktionsvoraussetzung die Variable `UR` an eine n-elementige Liste gebunden. Somit wird nach dem vorigen Lemma bei einem erfolgreichen Beweis des `append`-Literals die Variable `L` an eine $(n+1)$-elementige Liste gebunden. □

Schon dieses einfache Beispiel zeigt, daß diese Möglichkeit (exakte Beweise) auch nicht praktikabel ist, da die Durchführung solcher Beweise schwierig (Welche Hilfsaussagen werden benötigt? Wie sieht das Induktionsschema aus?) und *automatisch* im allgemeinen nicht möglich ist.

Aus diesen Gründen muß man davon absehen, die interessierenden Eigenschaften exakt herzuleiten. Als Alternative versucht man, die Eigenschaften nur soweit zu approximieren, daß sie in jedem Fall (effizient) berechnet werden können. Ein Rahmen für eine solche Vorgehensweise wurde in [CC77] unter dem Namen „abstrakte Interpretation“ eingeführt. Diese Technik wurde in vielen weiteren Arbeiten verbessert und für spezielle Anwendungen eingesetzt. Den „state of the art“ kann man in [AH87] nachlesen. Wir geben im folgenden nur die grundlegenden Ideen der abstrakten Interpretation in Bezug auf Prolog an.

6.2 Abstrakte Interpretation logischer Programme

Die Idee der abstrakten Interpretation ist, zum Berechnen der interessierenden Eigenschaften das Programm nicht mit konkreten Werten auszuführen (Terminierungsproblem), sondern stattdessen mit „abstrakten Werten“ zu rechnen. Wenn die Menge der abstrakten Werte endlich ist, kann (unter bestimmten zusätzlichen Voraussetzungen) garantiert werden, daß diese abstrakte Berechnung (Interpretation) auch immer endlich ist. Die abstrakten Werte sind so zu wählen, daß aus dem Ergebnis der abstrakten Interpretation Rückschlüsse auf die interessierenden Eigenschaften gezogen werden können.

Beispiel: Zur Analyse der Modi ist es wichtig zu wissen, ob eine Variable an einen Grundterm oder an eine freie Variable gebunden ist. Daher können hierfür die abstrakten Werte „ground“, „free“ und „any“ gewählt werden. Jeder Grundterm wird also zu „ground“, jede freie Variable zu „free“ und jeder andere Term zu „any“ abstrahiert. Wenn wir einen abstrakten Interpreter implementieren wollen, der mit diesen Werten rechnet, so sind folgende Probleme zu lösen:

1. Wie kann garantiert werden, daß der abstrakte Interpreter immer terminiert?
2. In welchem Zusammenhang stehen die berechneten abstrakten Werte mit den konkreten Werten beim Programmablauf? (D.h. welche Schlußfolgerungen kann man aus der abstrakten Interpretation sicher ziehen?)

Da wir im Rahmen dieser Vorlesung diese Details nicht darstellen können (dies würde den Inhalt einer eigenen Vorlesung ausmachen, vgl. Vorlesung „Datenflußanalyse“), werden wir im folgenden nur einen Überblick über eine mögliche Lösung geben. Vorschläge für Rahmen zur abstrakten Interpretation von Prolog-Programmen und die zugehörigen formalen Grundlagen findet man u.a. in [JS87] [Mel87] [BJCD87] [Deb88], [Nil88], [Nil90a] und [Bru91]. In [WHD88] werden die praktischen Einsatzmöglichkeiten dieser Techniken diskutiert. [TL92] enthält Hinweise zur effizienten Implementierung abstrakter Interpretationstechniken basierend auf ähnlichen Übersetzungstechniken, wie wir sie bisher kennengelernt haben. Im folgenden halten wir uns an die Darstellung in [Nil88].

Was sind nun i.a. die „interessierenden Programmeigenschaften“, die man schon zur Compilezeit kennen möchte? In imperativen Programmiersprachen sind dies die Werte der Variablen, in logischen Programmiersprachen entspricht dies den *Substitutionen*, die zur Laufzeit auftreten. Zur Optimierung von logischen Programmen möchte man also wissen, welche Substitutionen an bestimmten Stellen im Programm zur Laufzeit auftreten können.

Beispiel: Wenn im Programm jeder Aufruf von `reverse(L1,L2)` zur Laufzeit nur mit Substitutionen aus der Menge

$$\{\sigma \mid \sigma(\texttt{L1}) \text{ istGrundterm}, \ \sigma(\texttt{L2}) \in Var\}$$

erfolgt, dann hat `reverse` den Modus `(ground,free)`.

Daher sind wir daran interessiert, an jeder Stelle im Programm die Menge aller dort zur Laufzeit auftretenden Substitutionen zu wissen. Solche Mengen von Substitutionen nennen wir auch **zusammengefaßte Substitutionen** („summarizing substitutions“). Zur Präzisierung der nachfolgenden Methode zur Bestimmung solcher zusammengefaßter Substitutionen definieren wir die folgenden Begriffe:

- Operational können Klauselrümpfe und die Anfrage an das Programm als Folge von Prädikataufrufen gesehen werden. Bei jedem Aufruf (Literal) unterscheiden wir zwei sogenannte *Programmpunkte*: Den **Aufrufspunkt** vor und den **Erfolgspunkt** nach dem Literal.
- Der linke bzw. rechte Programmpunkt in einer Klausel heißt **Eintritts-** bzw. **Austrittspunkt** der Klausel.
- Zu jedem Programmpunkt assoziieren wir eine Menge von Substitutionen: $\theta_{call(L)}$ bzw. $\theta_{succ(L)}$ bezeichnen die zum Aufrufs- bzw. Erfolgspunkt eines Literals L gehörigen zusammengefaßten Substitutionen, und $\theta_{entry(C)}$ bzw. $\theta_{exit(C)}$ diejenigen zum Eintritts- bzw. Austrittpunkt einer Klausel C gehörigen.
- Analysiert wird ein Programm bzgl. einer Menge von möglichen Anfragen. Der Einfachheit halber nehmen wir an, daß dies durch eine Anfrage der Form

  $$? - p(X_1, \ldots, X_n)$$

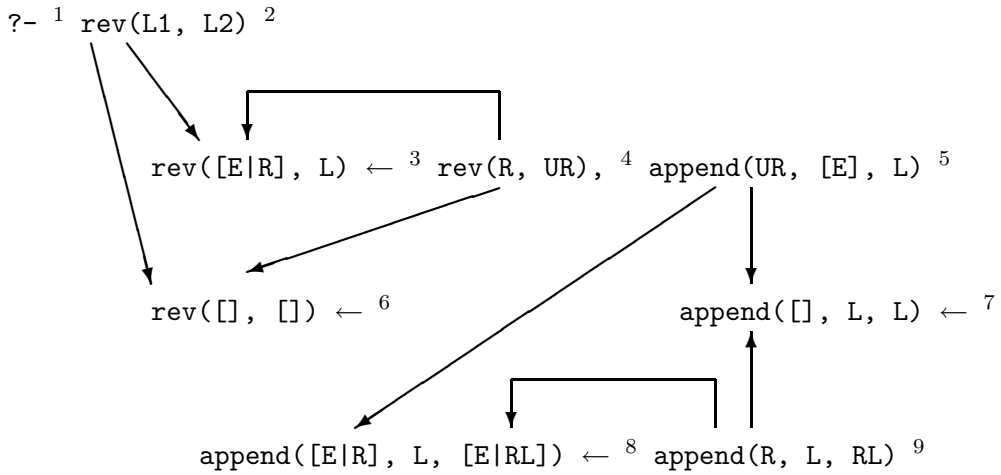
  und einer zugehörigen Menge von Substitutionen spezifiziert ist.

- Die Abhängigkeiten in einem Programm werden durch einen **Verbindungsgraphen** dargestellt, der Literale mit passenden Klauseln verbindet: Ist L ein Literal im Rumpf einer Klausel oder in der Anfrage und $C = H \leftarrow B$ eine Klausel, dann existiert eine Kante von L nach C g.d.w. H und L unifizierbar sind. Die Existenz einer solchen Kante wird mit $L \rightarrow C$ bezeichnet.

Beispiel: Programm `rev` zur Umkehrung einer Liste:

```
rev([], []) ←
rev([E|R], L) ← rev(R, UR), append(UR, [E], L)
append([], L, L) ←
append([E|R], L, [E|RL]) ← append(R, L, RL)
?- rev(L1, L2)
```

Der zugehörige Verbindungsgraph ist:



Da die Anzahl der Programmpunkte endlich ist, können wir sie durchnumerieren. Während der (abstrakten) Interpretation eines Programms werden die zusammengefaßten Substitutionen für diese berechnet und zu diesem Zweck werden diese markiert. Der Algorithmus hierfür lautet wie folgt:

Algorithmus: (Abstrakte) Interpretation eines logischen Programms

<Markiere den Aufrufspunkt der Anfrage. Alle anderen Programmpunkte sind unmarkiert.>

<Setze $\theta_{call(L)}$ (wobei L die Anfrage ist) auf die vorgegebene Menge der Anfragesubstitutionen. Setze die Menge der Substitutionen aller anderen Programmpunkte auf leer.>

WHILE < es existiert ein markierter Programmpunkt p >
DO < Lösche Markierung an p >
IF < p ist der Aufrufspunkt eines Literals L >
THEN FOR < alle Klauseln C mit $L \rightarrow C$ >
DO $\theta_{entry(C)} := \theta_{entry(C)} \cup (mgu(\theta_{call(L)}(L), \sigma(Head(C))) \circ \sigma) \mid_{vars(C)}$
{ wobei σ eine neue Variablenumbenennung ist }

$<$ markiere Eintrittspunkt von C, falls sich $\theta_{entry(C)}$ geändert hat $>$

OD

FI

IF $<$ p ist der Austrittspunkt einer Klausel C $>$

THEN FOR $<$ alle Literale L (in Klausel C') mit $L \rightarrow C$ $>$

DO $\theta_{succ(L)} := \theta_{succ(L)} \cup (mgu(\theta_{exit(C)}(Head(C)), \theta_{call(L)}(L)) \circ \theta_{call(L)}) \mid_{vars(C')}$

$<$markiere Erfolgspunkt von L, falls sich $\theta_{succ(L)}$ geändert hat$>$

OD

FI

OD

Hierbei gelten die folgenden Konventionen:

- $Head(C)$ bezeichnet den Kopf einer Klausel C.
- $\sigma|_V$ bezeichnet die Einschränkung der Substitution σ auf die Variablenmenge V.
- Die Anwendung einer Menge von Substitutionen auf ein Literal (wie $\theta_{call(L)}(L)$) ergibt sich aus der Vereinigung der Anwendung jeder einzelnen Substitution.

Dieser Algorithmus berechnet also iterativ Mengen von Substitutionen an den Programmpunkten solange, bis ein Fixpunkt bei der Berechnung erreicht ist. Hierbei werden i.a. nicht die exakten Substitutionsmengen berechnet, die zur Laufzeit auftreten können (dies ist natürlich auch nicht das Ziel der abstrakten Interpretation), sondern Obermengen hiervon. Zur praktischen Anwendung dieses Algorithmus' müssen wir endliche Beschreibungen der (potentiell unendlichen) Mengen von Substitutionen finden, die zudem so gewählt sind, daß

- hieraus die interessierenden Eigenschaften ableitbar sind,
- der Algorithmus immer terminiert, und
- die Ergebnisse korrekt sind.

Zu diesem Zweck gehen wir von der Menge der (konkreten) Substitutionen über zu einem **Verband von abstrakten Substitutionen**. Der Zusammenhang zwischen konkreten und abstrakten Substitutionen wird durch zwei Funktionen beschrieben:

Abstraktionsfunktion: $\alpha: 2^{\{konkrete\ Substitutionen\}} \rightarrow \{abstrakte\ Substitutionen\}$

Konkretisierungsfunktion: $\gamma: \{abstrakte\ Substitutionen\} \rightarrow 2^{\{konkrete\ Substitutionen\}}$

Die Abstraktionsfunktion α ordnet jeder Teilmenge der konkreten Substitutionen eine abstrakte Substitution zu, wogegen die Konkretisierungsfunktion γ die abstrakten Substitutionen „konkret interpretiert". Mit α wird die zu der Anfrage gegebene Menge konkreter Substitutionen in eine abstrakte Substitution übersetzt, dann rechnet der Algorithmus auf der Basis abstrakter Substitutionen einen Fixpunkt aus und anschließend kann das Ergebnis mittels γ konkret interpretiert werden.

Der Algorithmus arbeitet also nur mit abstrakten Substitutionen. Damit er terminiert, sollen die abstrakten Substitutionen einen Verband mit einer partiellen Ordnung $\sqsubseteq$ bilden, in dem keine unendlichen Ketten existieren und bei dem für zwei Elemente φ_1 und φ_2 immer eine kleinste obere Schranke $\varphi_1 \sqcup \varphi_2$ definiert ist. Wenn im obigen Algorithmus die Vereinigung $\cup$ von Mengen konkreter Substitutionen als $\sqcup$ interpretiert wird, ist die Terminierung garantiert (da jede aufsteigende Kette endlich ist). Zur Korrektheit sollen folgende Bedingungen erfüllt sein:

- Monotonie: $\sigma_1 \subseteq \sigma_2$ (σ_i Menge konkreter Substitutionen) impliziert $\alpha(\sigma_1) \sqsubseteq \alpha(\sigma_2)$, und $\varphi_1 \sqsubseteq \varphi_2$ (φ_i abstrakte Substitution) impliziert $\gamma(\varphi_1) \subseteq \gamma(\varphi_2)$
- $\alpha(\gamma(\varphi)) = \varphi$ (φ abstrakte Substitution)
- $\gamma(\alpha(\sigma)) \supseteq \sigma$ (σ Menge konkreter Substitutionen)

Im letzten Punkt ist nur Enthaltensein und keine Gleichheit gefordert, da eine Abstraktion immer einen Informationsverlust zur Folge hat.

Damit der Algorithmus mit den abstrakten Substitutionen rechnen kann, müssen die dort vorkommenden Operationen auf konkreten Substitutionen (Vereinigung, Komposition, Applikation und Unifikation) auf abstrakte Substitutionen übertragen werden. Zur Korrektheit ist folgendes zu beachten:

- Die *Vereinigung* $\cup$ wird auf abstrakten Substitutionen durch $\sqcup$ interpretiert.
- Die *Komposition* zweier abstrakter Substitutionen φ_1, φ_2 muß die folgende Eigenschaft erfüllen: Falls $\sigma_1 \in \gamma(\varphi_1), \sigma_2 \in \gamma(\varphi_2)$, dann $\sigma_1\sigma_2 \in \gamma(\varphi_1\varphi_2)$.
- Für die *Applikation* einer abstrakten Substitution φ auf ein Literal L soll gelten: Falls $\sigma \in \gamma(\varphi)$, dann $\sigma(L) \in \gamma(\varphi(L))$.
- Für die *abstrakte Unifikation* soll gelten: Falls $t_1 \in \gamma(\varphi_1(s_1))$, $t_2 \in \gamma(\varphi_2(s_2))$ und der mgu von t_1 und t_2 ist σ, dann soll gelten: $\sigma \in \gamma(mgu(\varphi_1(s_1), \varphi_2(s_2)))$.

Als Beispiel für eine Anwendung dieses Analyserahmens wollen wir Informationen darüber ableiten, ob Variablen zur Laufzeit immer an bestimmten Stellen an Grundterme gebunden sind **(„groundness analysis“)**. Die interessierenden Eigenschaften können charakterisiert werden durch die Menge der Variablen, die an Grundterme gebunden sind. Somit wird eine abstrakte Substitution dargestellt als Menge von Variablen, die an Grundterme gebunden sind. Da die abstrakten Substitutionen immer nur auf den Variablen einer Klausel definiert sind, sind diese Mengen immer endlich.

Die abstrakten Substitutionen für eine Klausel C sind definiert als Verband mit Elementen aus $2^{Vars(C)}$ (Teilmengen von Variablen der Klausel C), wobei gilt:

1. $A \sqsubseteq B$ g.d.w. $A \supseteq B$ für alle $A, B \subseteq Vars(C)$
2. Kleinste obere Schranke $\sqcup$: $A \sqcup B = A \cap B$ für alle $A, B \subseteq Vars(C)$

Bezgüglich der Ordnung $\sqsubseteq$ ist also $Vars(C)$ das kleinste und $\emptyset$ das größte Element im Verband. Intuitiv steht $\{X_1, \ldots, X_n\}$ für die Menge aller Substitutionen, die die Variablen $X_1, \ldots, X_n$ auf Grundterme abbilden. Formal:

$$\textbf{Abstraktion:} \quad \alpha(\Sigma) = \{X \in Vars(C) \mid \sigma(X) \text{ ist Grundterm } \forall \sigma \in \Sigma\}$$

$$\textbf{Konkretisierung:} \quad \gamma(V) = \{\sigma \text{ Substitution} \mid \sigma(X) \text{ ist Grundterm } \forall X \in V\}$$

Behauptung: α und γ sind monoton (Beweis zur Übung).

Zur Anwendung des Interpretationsschemas müssen wir die vorkommenden Operationen auf abstrakten Substitutionen definieren:

Die Vereinigung $\cup$ ist die kleinste obere Schranke $\sqcup$, die Komposition ist die Vereinigung der Variablenmengen, Applikation und Unifikation kann man auch einfach definieren (Übung).

Betrachten wir das o.a. `rev`-Programm mit der abstrakten Substitution {`L1`} für die Anfrage, d.h. bei Aufruf des Programms ist das erste Argument von `rev` ein Grundterm. Dann wird der Programmpunkt 1 mit {`L1`} und alle anderen Punkte mit $Vars(C)$ markiert, wobei C die Klausel ist, in der der entsprechende Programmpunkt vorkommt. Die iterative Berechnung der abstrakten Substitutionen führt zu folgendem Ergebnis:

Programmpunkt	**abstrakte Substitution**
1	{`L1`}
2	{`L1`, `L2`}
3	{`E`, `R`}
4	{`E`, `R`, `UR`}
5	{`E`, `R`, `UR`, `L`}
6	{}
7	{`L`}
8	{`E`, `R`, `L`}
9	{`E`, `R`, `L`, `RL`}

Hieraus können wir folgende Schlußfolgerungen ziehen:

- Wenn die Anfrage beweisbar ist, dann ist `L2` an einen Grundterm gebunden (Punkt 2).
- Jeder Aufruf von `rev` erfolgt zur Laufzeit mit einem Grundterm als erstem Argument (Punkte 1 und 3).
- Bei jedem Aufruf von `append` sind die ersten beiden Argumente an Grundterme gebunden (Punkte 4 und 8).

Mit diesen Informationen kann der Compiler das Indizierungsschema für `rev` und `append` vereinfachen (s.o.).

6.3 Anwendung abstrakter Interpretationstechniken zur Implementierung

In der Einleitung zu diesem Kapitel haben wir schon darauf hingewiesen, daß eine wichtige Motivation zur Entwicklung abstrakter Interpretationstechniken die Möglichkeit ist, logische Programme in effizienter ausführbaren Code zu übersetzen. Dies wurde auch schon vor einigen Jahren von Mellish [Mel85] an Beispielen demonstriert. Dennoch stellt sich die Frage, wie man in einem echten Prolog-Compiler Laufzeitinformation herleiten und bei der Übersetzung einsetzen kann. In diesem Abschnitt wollen wir kurz die Erfahrungen aus zwei Compiler-Projekten skizzieren, bei denen abstrakte Interpretationstechniken eingesetzt wurden.

Peter Van Roy hat sich in seiner Dissertation [VR90] die Frage gestellt, ob logische Programme genau so schnell ausgeführt werden können wie imperative Programme. Motivation für diese Fragestellung ist die häufig vertretene Ansicht, daß logische Programmiersprachen bezüglich Ausführungsgeschwindigkeit nie mit imperativen Programmiersprachen konkurrieren können, da logische Programmiersprachen aufgrund ihrer höheren Ausdrucksmächtigkeit nicht so effizient implementiert werden können. Diese Auffassung versucht Peter Van Roy zu widerlegen, indem er eine Prolog-Implementierung (Aquarius-Prolog) entwickelt, deren Ausführungsgeschwindigkeit mit imperativen Sprachen (hier: C) vergleichbar ist. Dieser Implementierung liegen die folgenden Überlegungen zugrunde:

- Logische Programmiersprachen sind zwar ausdrucksmächtiger als imperative Sprachen, allerdings wird häufig (bzw. in großen Programmteilen) diese Ausdrucksmächtigkeit nicht ausgenutzt. Wenn diese Ausdrucksmächtigkeit nicht genutzt wird, sollte auch der Compiler einen effizienten Code erzeugen.

- Die Analyse der Programmteile, in denen die besonderen Eigenschaften der logischen Programmierung (logische Variablen und Unifikation, Lösungssuche durch Backtracking) weitgehend nicht benutzt werden, erfolgt mittels abstrakter Interpretationstechniken. Dies ist gegenüber expliziten Programmannotationen durch den Programmierer sicherer (Annotationen könnten auch falsch gesetzt werden) und für den Programmierer einfacher.

- Die WAM ist ein guter Ansatz zur Implementierung logischer Programmiersprachen. Allerdings sind die Instruktionen zu komplex für eine effiziente Implementierung auf einer konkreten Standardhardware. Dies liegt insbesondere daran, daß die Instruktionen für die allgemeine logische Programmierung ausgelegt sind, jedoch viele praktische Programme diesen Umfang nicht ausnutzen. Daher muß der Compiler als Zielmaschine nicht die WAM nehmen, sondern eine speziellere Maschine, die besser auf die konkrete Hardware angepaßt ist.

- Der Compiler beinhaltet somit einen abstrakten Interpreter, der genau die Laufzeitinformation bereitstellt, mit der der Codegenerator dann die speziellen Maschineninstruktionen erzeugen kann.

Eine genaue Beschreibung dieses Compilers zusammen mit den Laufzeitergebnissen befindet sich in [VR90] (in [VR89] ist eine wichtige Teiltechnik dargestellt, nämlich eine Verfeinerung der WAM-Instruktionen zur Unifikation). Van Roy hat seine Implementierung mit einem schnellen kommerziellen

Prolog-System basierend auf der WAM (Quintus-Prolog) verglichen. Dabei ist sein Aquarius-System im Durchschnitt etwa fünfmal schneller! Außerdem hat er sein System mit einem optimierenden C-Compiler verglichen (dies war ja der Ausgangspunkt der Fragestellung). Insbesondere bei rekursiven arithmetischen Funktionen (Fakultät, Fibonacci-Zahlen) ist das Aquarius-System schneller als der optimierende C-Compiler! Dies hat u.a. folgende Ursachen:

1. Bei der Implementierung von Prolog wird die Rekursion besonders gut implementiert (tail recursion optimization), weil hier jede Schleife durch Rekursion ausgedrückt werden muß. In C-Compilern werden rekursive Funktionen nicht so gut implementiert, da hier die Programmierer mehr Schleifen benutzen.

2. Das Aquarius-System versucht deterministische Prädikate in möglichst vielen Fällen auch deterministisch zu implementieren. Als Beispiel betrachten wir das Prädikat `max` zur Definition des maximalen Wertes zweier Zahlen:

   ```
   max(X,Y,Z) ← X >= Y,  Z = X
   max(X,Y,Z) ← X < Y,  Z = Y
   ```

   Bei unserem bisherigen Implementierungsschema (vgl. Kapitel 4) wird beim Beweis des Literals `max(3,5,V)` zunächst ein Backtrackpunkt aufgebaut, dann die erste Klausel ausprobiert, und anschließend durch den Fehlschlag der Bedingung `3 >= 5` ein Backtrackschritt durchgeführt bevor die zweite Klausel erfolgreich angewendet werden kann. Durch eine Analyse der ersten Bedingungen in den Klauseln kann jedoch das Prädikat in einen Code übersetzt werden, der ungefähr der folgenden Definition entspricht:

   ```
   max(X,Y,Z) ← if X >= Y then Z = X else Z = Y fi
   ```

   Bei dieser optimierten Übersetzung wird dagegen kein Backtrackpunkt aufgebaut und `max` wird immer deterministisch abgearbeitet. Zu weiteren Details siehe [VR90].

Andrew Taylor hat sich ebenfalls mit der Anwendung abstrakter Interpretationstechniken zur besseren Übersetzung logischer Programme beschäftigt [Tay89] [Tay90]. Ausgangspunkt ist hier ebenfalls die Überlegung, daß der WAM-Code in vielen Fällen zu allgemein und daher ineffizient ist. Betrachten wir z.B. die erste Klausel für das bekannte `append`-Prädikat:

```
append([], L, L) ←
```

Die Übersetzung in WAM-Instruktionen scheint auf den ersten Blick sehr gut zu sein:

```
get_nil A1
get_value A2, A3
proceed
```

Bedenkt man jedoch, was sich hinter diesen Instruktionen verbirgt, so sieht man sofort, daß die Ausführung einigen Aufwand erfordert. Es werden nämlich ungefähr die folgenden Aktionen durchgeführt:

⟨Dereferenziere `A1`⟩
IF ⟨`A1` gebunden⟩
THEN IF `A1 = []`
THEN ⟨Unifiziere `A2` und `A3`⟩
ELSE `fail`
FI
ELSE ⟨Binde `A1` an Konstante `[]`⟩
IF ⟨Muß Bindung von `A1` auf dem Trail vermerkt werden?⟩
THEN `trail(A1)`
FI
⟨Unifiziere `A2` und `A3`⟩
FI

Wie man sieht, werden neben einigen einfachen Instruktionen auch noch komplexe Prozeduren aufgerufen, wie die Dereferenzierung von `A1` und die volle Unifikation von `A2` und `A3`. Taylor schlägt daher vor, durch eine Analyse des Programms Informationen darüber abzuleiten, mit welchen Klassen von Werten die Prädikate aufgerufen werden. Wenn man z.B. abgeleitet hat, daß bei jedem Aufruf von `append` die ersten beiden Argumente Grundterme und das dritte Argument eine freie Variable ist, wobei das erste Argument immer voll derefenziert ist und eine Bindung des dritten Argumentes nicht auf dem Trail protokolliert werden muß (dies kommt in realen Prolog-Programmen tatsächlich sehr häufig vor), dann kann der Code wie folgt vereinfacht werden:

IF `A1 = []`
THEN ⟨Binde `A3` an `A2`⟩
ELSE `fail`
FI

Um diese Laufzeitinformation abzuleiten, benötigt man allerdings einen komplexen abstrakten Wertebereich. Taylor schlägt dazu einen abstrakten Wertebereich vor, der die folgenden Elemente entält:

```
nil       atom      integer    float      number     constant
ground    free      notvar     unknown    term       list
```

Dabei sind die einzelnen Elemente noch parametrisiert mit weiteren Informationen, wie z.B. die Länge der möglichen Zeigerketten, die Notwendigkeit Variablenbindungen auf dem Trail zu protokollieren etc. Mit Hilfe einer abstrakten Interpretation über diesem Wertebereich werden in den meisten Fällen brauchbare Informationen abgeleitet. Z.B. werden bei verschiedenen real eingesetzten Prolog-Programmen in 70-90% der Fälle die Modi + bzw. − für Prädikatargumente abgeleitet, so daß die entsprechenden Abfragen im erzeugten Code entfallen können.

Mit Hilfe der errechneten Laufzeitinformation übersetzt der Compiler Prolog-Programme in entsprechende Maschinenprogramme für eine RISC-Architektur (MIPS M12). Der Standardbenchmark “nai-

ve reverse" (vgl. Kapitel 4.6) wird mit etwa 2 megaLIPS abgearbeitet. Diese Implementierung ist im Durchschnitt 24-mal schneller als ein Prolog-Compiler ohne abstrakte Interpretation (auf gleicher Hardware) und immerhin 2,5-mal schneller als eine Spezialhardware für Prolog. Und auch der erzeugte Code ist in der Regel weniger als halb so groß wie bei anderen Prolog-Compilern, obwohl bei Taylors System der Code in manchen Fällen teilweise dupliziert wird (z.B. wenn ein Prädikat in zwei verschiedenen Modi aufgerufen wird, dann wird für jeden Modus ein separater Code erzeugt).

Diese Ergebnisse zeigen deutlich, daß man durch gute Übersetzungstechniken die Effizienz der ausgeführten Programme wesentlich verbessern kann. Bei deklarativen Programmiersprachen ist der Einsatz abstrakter Interpretationstechniken besonders lohnenswert, da man dann in vielen Fällen den zunächst allgemein gehaltenen Code gut spezialisieren kann. Allerdings darf man einen Nachteil abstrakter Interpretationstechniken nicht übersehen: die abstrakte Interpretation eines Programms kostet Zeit, und diese ist im allgemeinen nicht linear abhängig von der Programmgröße, da die Fixpunktberechnung über dem abstrakten Wertebereich im Extremfall viele Durchläufe über das Programm nötig macht. Debray [Deb91] hat die Komplexität verschiedener Datenflußanalyseaufgaben untersucht und dabei bewiesen, daß die Ableitung der meisten interessanten Informationen eine polynomielle oder gar exponentielle Laufzeit erfordert. Dies ist allerdings nur im schlechtesten Fall so. Bei vielen praktischen Programmen terminiert der abstrakte Interpreter wesentlich schneller. Dennoch ist eine gute Implementierung von abstrakten Interpretern eine wichtige Aufgabe für die Zukunft, damit abstrakte Interpretationstechniken auch in kommerziellen Prolog-Systemen angewendet werden können.

Literaturverzeichnis

[AB87] B. Arbab and D.M. Berry. Operational and Denotational Semantics of Prolog. *Journal of Logic Programming (4)*, pp. 309–329, 1987.

[ACHS88] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage Collection for Prolog Based on WAM. *Communications of the ACM*, Vol. 31, No. 6, pp. 719–741, 1988.

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[AK91] H. Aït-Kaci. *Warren's Abstract Machine*. MIT Press, 1991.

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1985.

[BBCT86] K.A. Bowen, K.A. Buettner, I. Cicekli, and A.K. Turk. The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler. In *Proc. Third International Conference on Logic Programming (London)*, pp. 650–656. Springer LNCS 225, 1986.

[BC83] M. Bidoit and J. Corbin. A Rehabilitation of Robinson's Unification Algorithm. In *Proc. IFIP '83*, pp. 909–914. North-Holland, 1983.

[BJ88] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inferencing. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 669–683, 1988.

[BJCD87] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: towards the global optimization of Prolog programs. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 192–204, San Francisco, 1987.

[BM72] R.S. Boyer and J.S. Moore. The sharing of structure in theorem proving programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pp. 101–116, 1972.

[BM86] J. Barklund and H. Millroth. Garbage Cut for Garbage Collection of Iterative Prolog Programs. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 276–283, Salt Lake City, 1986.

[Bör90a] E. Börger. A Logical Operational Semantics of Full Prolog. Part I: Selection Core and Control. In *Proc. CSL'89*, pp. 36–64. Springer LNCS 440, 1990.

[Bör90b] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database manipulations. In *Mathematical Foundations of Computer Science '90*, pp. 1–14. Springer LNCS 452, 1990.

[Bru82] M. Bruynooghe. The Memory Management of PROLOG Implementations. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pp. 83–98. Academic Press, 1982.

[Bru84] M. Bruynooghe. Garbage collection in Prolog interpreters. In J.A. Campbell, editor, *Implementations of Prolog*, pp. 259–267. Ellis Horwood, 1984.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming (10)*, pp. 91–124, 1991.

[Bue86] K.A. Buettner. Fast Decompilation of Compiled Prolog Clauses. In *Proc. Third International Conference on Logic Programming (London)*, pp. 663–670. Springer LNCS 225, 1986.

[Cam84] J.A. Campbell. *Implementations of Prolog*. Ellis Horwood, 1984.

[Car87] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 40–58. MIT Press, 1987.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

[Clo85a] W.F. Clocksin. Design and Simulation of a Sequential Prolog Machine. *New Generation Computing*, Vol. 3, No. 1, pp. 101–120, 1985.

[Clo85b] W.F. Clocksin. Implementation Techniques for Prolog Databases. *Software - Practice and Experience*, Vol. 15, No. 7, pp. 669–675, 1985.

[Coh85] J. Cohen. Describing Prolog by its Interpretation and Compilation. *Communications of the ACM*, Vol. 28, No. 12, pp. 1311–1324, 1985.

[Coh88] J. Cohen. A View of the Origins and Development of Prolog. *Communications of the ACM*, Vol. 31, No. 1, pp. 26–36, 1988.

[Deb86] S.K. Debray. Register Allocation in a Prolog Machine. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 267–275, Salt Lake City, 1986.

[Deb88] S.K. Debray. Efficient Dataflow Analysis of Logic Programs. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pp. 260–273, San Diego, 1988.

[Deb91] S.K. Debray. On the Complexity of Dataflow Analysis of Logic Programs. Report TR 91-27, Univ. of Arizona, 1991.

[DF87] P. Deransart and G. Ferrand. An Operational Formal Definition of PROLOG. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 162–172, San Francisco, 1987.

[DM88] S.K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming (5)*, pp. 61–91, 1988.

[DW88] S.K. Debray and D.S. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming (5)*, pp. 207–229, 1988.

[DW89] S.K. Debray and D.S. Warren. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pp. 451–481, 1989.

[FA88] D.A. Fuller and S. Abramsky. Mixed Computation of Prolog Programs. *New Generation Computing*, Vol. 6, pp. 119–142, 1988.

[Fuc84] K. Fuchi. Logical Derivation of a Prolog Interpreter. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 229–234. ICOT, 1984.

[GLLO85] J. Gabriel, T. Lindholm, E.L. Lusk, and R.A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Technical Report ANL-84-84, Argonne National Laboratory, Argonne (Illinois), 1985.

[GM86] J.A. Goguen and J. Meseguer. Eqlog: Equality, Types, and Generic Modules for Logic Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 295–363. Prentice Hall, 1986.

[GMP86] J. Gee, S.W. Melvin, and Y.N. Patt. The Implementation of Prolog via VAX 8600 Microcode. In *Proc. 19th Annual Workshop on Microprogramming*, pp. 68–74, New York, 1986.

[Gre69] C.C. Green. *The Application of Theorem Proving Question-Answering Systems.* PhD thesis, Stanford University, 1969.

[Han87] M. Hanus. *Problemlösen mit Prolog.* Teubner, Stuttgart, 1987. 2. überarbeitete und erweiterte Auflage.

[Han88] M. Hanus. Formal Specification of a Prolog Compiler. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, pp. 273–282, Orléans, 1988. Springer LNCS 348.

[JDM88] G. Janssens, B. Demoen, and A. Marien. Improving the Register Allocation in WAM by Reordering Unification. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 1388–1402. MIT Press, 1988.

[JM84] N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 281–288, Atlantic City, 1984.

[JS87] N. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of PROLOG. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pp. 123–142. Ellis Horwood, 1987.

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, pp. 124–140. Springer LNCS 202, 1985.

[KC84] K.M. Kahn and M. Carlsson. The Compilation of Prolog Programs without the Use of a Prolog Compiler. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 348–355. ICOT, 1984.

[KK71] R. Kowalski and D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, Vol. 2, pp. 227–260, 1971.

[Klu88] F. Kluźniak. Compile Time Garbage Collection for Ground Prolog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 1490–1505, 1988.

[Kow74] R. Kowalski. Predicate Logic as Programming Language. In *Proc. IFIP '74*, pp. 569–574. North-Holland, 1974.

[Kow88] R. Kowalski. The Early Years of Logic Programming. *Communications of the ACM*, Vol. 31, No. 1, pp. 38–43, 1988.

[KS85] F. Kluźniak and S. Szpakowicz. *Prolog for Programmers*. Academic Press, London, 1985.

[KSKH85] M. Kishimoto, T. Shinogi, Y. Kimura, and A. Hattori. Design and Evaluation of a Prolog Compiler. In *Logic Programming '85 (Tokyo)*, pp. 192–203. Springer LNCS 221, 1985.

[KTW+86] Y. Kaneda, N. Tamura, K. Wada, H. Matsuda, S. Kuo, and S. Maekawa. Sequential Prolog Machine PEK. *New Generation Computing*, Vol. 4, No. 1, pp. 51–66, 1986.

[Kur86] P. Kursawe. How to invent a Prolog machine. In *Proc. Third International Conference on Logic Programming (London)*, pp. 134–148. Springer LNCS 225, 1986.

[KYAB88] K. Kurosawa, S. Yamaguchi, S. Abe, and T. Bandoh. Instruction Architecture for a High Performance Integrated Prolog Processor IPP. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 1506–1530, 1988.

[LBD+87] N. Lindenberg, A. Bockmayr, R. Dietrich, P. Kursawe, B. Neidecker, C. Scharnhorst, and I. Varsek. KA-Prolog: Sprachdefinition. Technical Report 5/87, Univ. Karlsruhe, 1987.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[LO87] T.G. Lindholm and R.A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 21–39. MIT Press, 1987.

[LS88] G. Levi and G. Sardu. Partial Evaluation of Metaprograms in a "Multiple Worlds" Logic Language. *New Generation Computing*, Vol. 6, pp. 227–247, 1988.

[Mel82] C.S. Mellish. An Alternative to Structure Sharing in the Implementation of a PROLOG Interpreter. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pp. 99–106. Academic Press, 1982.

[Mel85] C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming (1)*, pp. 43–66, 1985.

[Mel87] C.S. Mellish. Abstract interpretation of PROLOG programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pp. 181–198. Ellis Horwood, 1987.

[Mil89] J.W. Mills. A High-Performance Low Risc Machine for Logic Programming. *Journal of Logic Programming (6)*, pp. 179–212, 1989.

[MJMB89] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 33–47. MIT Press, 1989.

[MM82] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.

[Mos86] C. Moss. CUT & PASTE - defining the impure Primitives of Prolog. In *Proc. Third International Conference on Logic Programming (London)*, pp. 686–694. Springer LNCS 225, 1986.

[MW88] D. Maier and D.S. Warren. *Computing with Logic - Logic Programming with Prolog.* Benjamin/Cummings, 1988.

[Nei86] B. Neidecker. KA-Prolog. KAP-Maschine: Maschinenmodell und Instruktionssatz. Technical Report 19/86, Univ. Karlsruhe, 1986.

[Nil84] J.F. Nilsson. Formal Vienna-Definition-Method models of Prolog. In J.A. Campbell, editor, *Implementations of Prolog*, pp. 281–308. Ellis Horwood, 1984.

[Nil88] U. Nilsson. Towards a Framework for the Abstract Interpretation of Logic Programs. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, pp. 68–82, Orléans, 1988. Springer LNCS 348.

[Nil90a] U. Nilsson. Systematic Semantic Approximations of Logic Programs. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 293–306. Springer LNCS 456, 1990.

[Nil90b] U. Nilsson. Towards a Methodology for the Design of Abstract Machines of Logic Programming Languages. Report LiTH-IDA-R-90-12, Univ. Linköping, 1990.

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

[PBW85] E. Pittomvils, M. Bruynooghe, and Y.D. Willems. Towards a Real-Time Garbage Collector for Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 185–198, Boston, 1985.

[Per87] F. Pereira. *C-Prolog User's Manual, Version 1.5.* University of Edinburgh, 1987.

[Pla84] D. Plaisted. The occur-check problem in Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 272–280, Atlantic City, 1984.

[Pro87] Projektgruppe ProCom. Zwischen- und Abschlußbericht der Projektgruppe ProCom (Prolog Compiler). Univ. Dortmund, 1987.

[PW78] M.S. Paterson and M.N. Wegman. Linear Unification. *Journal of Computer and System Sciences*, Vol. 17, pp. 348–375, 1978.

[Red84] U.S. Reddy. Transformation of Logic Programs into Functional Programs. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 187–196, Atlantic City, 1984.

[Rob65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, Vol. 12, No. 1, pp. 23–41, 1965.

[Rou75] P. Roussel. PROLOG, Manuel de Reference et d'Utilisation. Groupe Intelligence Artificielle, Université Aix-Marseille II, 1975.

[SB89] L. Sterling and R.D. Beer. Metainterpreters for Expert System Construction. *Journal of Logic Programming (6)*, pp. 163–178, 1989.

[Som87] Z. Somogyi. A system of precise modes for logic programs. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 769–787. MIT Press, 1987.

[Søn86] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *Proc. European Symposium on Programming*, pp. 327–338. Springer LNCS 213, 1986.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Tay89] A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 48–60. MIT Press, 1989.

[Tay90] A. Taylor. LIPS on a MIPS: Results form a Prolog Compiler for a RISC. In *Proc. Seventh International Conference on Logic Programming*, pp. 174–185. MIT Press, 1990.

[TD87] H. Touati and A. Despain. An Empirical Study of the Warren Abstract Machine. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 114–124, San Francisco, 1987.

[TF86] A. Tackeuchi and K. Furukawa. Partial Evaluation of Prolog Programs and its Application to Meta Programming. In *Proc. IFIP '86*, pp. 415–420. North-Holland, 1986.

[TL92] J. Tan and I-P. Lin. Compiling Dataflow Analysis of Logic Programs. In *Proc. SIGPLAN'92 PLDI*, pp. 106–115, 1992.

[Tur86] A.K. Turk. Compiler Optimizations for the WAM. In *Proc. Third International Conference on Logic Programming (London)*, pp. 657–662. Springer LNCS 225, 1986.

[TW84] E. Tick and D.H.D. Warren. Towards a Pipelined Prolog Processor. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 29–40, Atlantic City, 1984.

[VD88] R. Venken and B. Demoen. A Partial Evaluation System for Prolog: some Practical Considerations. *New Generation Computing*, Vol. 6, pp. 279–290, 1988.

[vE84] M.H. van Emden. An interpreting algorithm for Prolog programs. In J.A. Campbell, editor, *Implementations of Prolog*, pp. 93–110. Ellis Horwood, 1984.

[Ven84] R. Venken. A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query-Optimisation. In *ECAI 84*, pp. 91–100. North-Holland, 1984.

[VR84] P. Van Roy. A Prolog Compiler for the PLM. Report No. UCB/CSD 84/203, Univ. of California, Berkeley, 1984.

[VR89] P. Van Roy. An Intermediate Language to Support Prolog's Unification. In *Proc. of the 1989 North American Conference on Logic Programming*, pp. 1148–1164. MIT Press, 1989.

[VR90] P.L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.

[VRDW87] P. Van Roy, B. Demoen, and Y.D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection, and determinism. In *Proc. of the TAPSOFT '87*, pp. 111–125. Springer LNCS 250, 1987.

[War77] D.H.D. Warren. Implementing PROLOG - Compiling Logic Programs. 1 and 2. D.A.I. Research Report No. 39 and 40, University of Edinburgh, 1977.

[War80] D.H.D. Warren. Logic Programming and Compiler Writing. *Software - Practice and Experience*, Vol. 10, pp. 97–125, 1980.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.

[War84] D.S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 198–202, Atlantic City, 1984.

[WHD88] R. Warren, M. Hermenegildo, and S.K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 684–699, 1988.

[WP84] M.J. Wise and D.M.W. Powers. Indexing PROLOG Clauses via Superimposed Code Words and Field Encoded Words. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 203–210, Atlantic City, 1984.