D I P L O M A R B E I T

P R O L O G - U E B E R S E T Z E R

ausgefuehrt am Institut fuer

P r a k t i s c h e I n f o r m a t i k

der Technischen Universitaet Wien

unter Anleitung von O. Prof. M. Brockhaus

und

Univ. Ass. Dipl. Ing. A. Krall

durch

Christian Pichler

1200 Wien, Wolfsaugasse 4/11

November 1984

Christian Pichler

1. Abstract

Die Programmiersprache Prolog basiert auf dem Kalkuel der Praedikatenlogik, welche auf die Teilmenge der Hornklauseln eingeschraenkt wurde. Es ist eine einfache, aber maechtige Programmiersprache, welche es erlaubt Probleme, die mittels Objekten und Beziehungen zwischen diesen Objekten beschrieben sind, zu loesen. Die grundlegende Berechnungsvorschrift dieser Sprache besteht in einem Prozess (unify, vereinigen), der versucht zwei beliebige Datenstrukturen (Terme) so anzugleichen (pattern matching), dass sie identisch werden.

In dieser Abhandlung soll speziell gezeigt werden, wie Prologklauseln in Instruktionen einer auf einer tiefen Ebene befindlichen Sprache einer hypothetischen Prologmaschine umzusetzen sind. Der derzeitig implementierte Uebersetzer , der selbst in Prolog geschrieben wurde, und als builtin Praedikat eines Prologinterpreters arbeitet, uebersetzt Prologklauseln in die Sprache der eben genannten Prologmaschine, welche dann mittels eines in C geschriebenen Interpreters emuliert wird.

2. Einleitung

2.1. Die Programmiersprache Prolog

Robert Kowalski war einer der ersten Vertreter des Programmierens in Logik, wobei die Hauptidee im Interpretieren von Hornklauseln besteht. Prolog ist ein Beispiel einer Programmiersprache die auf logischem Programmieren basiert. Es wurde von einer Gruppe, geleitet von Alain Colmerauer, an der Universitaet von Marseille vor ungefaehr 10 Jahren entworfen. Etliche andere Gruppen, speziell an der Universitaet von Edinburgh, entwickelten es weiter.

Prolog zeichnet sich durch eine sehr kompakte Syntax, einen kleinen fixen Kern, und einer einfachen Semantik, die mittels des logischen Kalkuels arbeitet, aus. Ein Prolog-Programmablauf besteht hauptsaechlich aus einer Folge von Prozeduraufrufen, welche abhaengig von den uebergebenen Parametern durchgefuehrt werden. Der Mechanismus des Prozeduraufrufs ist gegenueber konventionellen Programmiersprachen viel flexibler gestaltet. So koennen einer Prozedur nicht nur mehrere Parameter uebergeben werden, sondern sie kann auch mehr als einen Wert zurueckliefern, wobei vom Benutzer nicht bereits bei der Programmerstellung entschieden werden muss, welche Parameter Eingangs- und welche Ausgangsparameter sind.

Weiters kann eine Prozedur indeterministisch sein, d.h. sie kann mittels 'Backtracking' verschiedene Resultate lie-

fern. Diese Eigenschaft entspricht dem menschlichen Denkprozess weitaus besser als das Formulieren eines Problems mit den Mitteln die eine konventionelle, deterministische Sprache zur Verfuegung stellt.

2.2. Terminologie

Die verwendete Terminologie laesst sich am besten dadurch erklaeren, indem die Syntax eines Prologprogrammes in dieser Terminologie beschrieben wird. Hierbei wird in Klammer ((...)) jeweils die entsprechende englische Bezeichnung angefuegt.

Programm (program) ::= Prozedur | Programm Prozedur

Prozedur (procedure) ::= Klausel | Klausel Prozedur

Klausel (clause) ::= Kopf :- Rumpf. | Kopf.

Kopf (head) ::= Funktor(Argumente) | Atom

Argumente ::= Terme

Rumpf (body) ::= Praedikat | Praedikat , Rumpf

Praedikat (goal) ::= Funktor(Parameter) | Atom

Parameter ::= Terme

Terme ::= Term | Term, Terme

Term (term,literal) ::= Struktur | Variable | Konstante

Struktur (structure) ::= Funktor(Terme)

Konstante(constant) ::= Atom | Integer

Atom (atom)

Integer (integer)

Variable (variable)

Funktor (functor)

Argumente bzw. Parameter entsprechen den formalen bzw. aktuellen Parametern aus konventionellen Programmiersprachen, wie z.B. Pascal.
Die Stelligkeit einer Struktur, gibt die Anzahl ihrer Argumente an.
Prozedur wird eine Menge von Klauseln genannt, die als Kopf denselben Funktor (Atom) mit derselben Stelligkeit besizten.

2.3. Die Semantik von Prolog

Die Semantik eines Prolog Programmes erlaubt zwei verschiedene Sichtweisen:

deklarativ und prozedural

Bei der prozeduralen Sichtweise wird ein Programm so interpretiert, wie man es von konventionellen Programmiersprachen gewohnt ist. Ausschlaggebend fuer die Interpretation ist die dynamische Ablauffolge von Prozeduraufrufen.

Die deklarative Semantik erlaubt die Interpretation von logischen Regeln.

Ein Prolog Programm kann verstanden werden, indem jede Klausel als natuerlichsprachliche Aussage interpretiert wird. So kann eine Klausel der Form

```
A :- B,C,D.
```

wie folgt interpretiert werden.

Die Aussage A ist richtig, falls die Aussagen, B, C und D richtig sind.
Jede Klausel einer Prozedure muss auf dieselbe Art interpretiert werden.

Diese Art der Interpretation eines Prologprogrammes soll durch ein einfaches Beispiel naeher erklaert werden.

```
append(nil,Liste,Liste).
append(cons(T,Liste1),Liste2,cons(T,Liste3)) :-
    append(Liste1,Liste2,Liste3).
```

nil sei die leere Liste
cons(T,Liste) sei so zu interpretieren, dass 'Liste' der Rest einer Liste ist die mit 'T' beginnt.
append(Liste1,Liste2,Liste3) sei so zu interpretieren, dass 'Liste3' die Verknuepfung von 'Liste1' mit 'Liste2' ist.

Nun koennen die beiden Klauseln wie folgt gedeutet werden.

Die leere Liste verknuepft mit einer Liste liefert als Ergebnis die Liste selbst.

Eine Liste die mit 'T' beginnt und in 'Liste1' endet, verknuepft mit einer Liste 'Liste2' resultiert genau dann in eine Liste die mit 'T' beginnt und mit 'Liste3' endet, wenn die Verknuepfung von 'Liste1' mit 'Liste2' die Liste 'Liste3' liefert.

2.4. Einsatzbereiche fuer Prolog

Nach Kowalski (siehe: new generation computing, vol.1 no.1 1983) bringt logisches Programmieren eine Synthese in den derzeitigen Trends der sonst getrennten Bereiche des Softwareengineering, der Datenbanksysteme, der Rechnerarchitektur und Kuenstlichen Intelligenz, eingeschlossen des Verarbeitens von natuerlicher Sprache.

Prolog hat eine leicht zu verstehende deklarative Semantik, die auf einstufiger Praedikatenlogik basiert, was Prolog zu einer sehr leistungsfaehigen Spezifikationssprache macht. Ein Prologprogramm hat zusaetzlich zu seiner deklarativen Bedeutung auch eine prozedurale Bedeutung, d.h. es kann ausgefuehrt werden. Eine Systemspezifikation die in Prolog geschrieben wurde, kann also als erster Prototyp fuer ein spezifiziertes System benutzt werden. Prolog stellt etliche Hilfsmittel fuer Simulation und Fehlerkorrektur zur Verfuegung. Seine Applikation ist unabhaengig von Programm-Paradigmata (z.b. strukturiert, funktional, objekt-orientiert) und Entwurfsstrategien (top-down, bottom-up). Wie die Semantik von Prologklauseln kein Konzept fuer Eingans- und Ausgangsparameter vorsieht, gilt dies auch fuer die formalen Parameter einer Prozedur. Abhaengig von der aktuellen Unifizierung der Datenobjekte X und Y bei ihrer Loesung, kann ein Prologpraedikat p(X,Y) benutzt werden um die logische Existenz der logischen Beziehung 'p' zwischen zwei Termen X und Y zu testen. In einem anderen Kontext kann dies

eine Regel sein um X aus Y zu bilden oder umgekehrt, oder um alle X-Y Paare zu bilden. Dies ist eine sehr nuetzliche Eigenschaft zum Prototyping, da dies zu einer grossen Praegnanz und Einfachheit fuehrt, um logische Bloecke zu bilden, die von einem bestimmten Task benoetigt werden.

Beim Softwareengineering wird immer mehr Aufmerksamkeit der Wichtigkeit von Spezifikationen und Spezifikationssprachen gelegt. Symbolische Logik ist als der beste Formalismus fuer Problemspezifikation anerkannt. Studien ueber automatisches Beweisen im generellen und logische Programmierung im speziellen zeigen dass Spezifikationen mit Hilfe des Computer manipuliert, ausgefuehrt und korrigiert werden koennen. Die Programmerstellung geht in derselben Weise vor. Sie startet mit einem klaren, aber uneffizienten Programm, das in einer funktionalen Programmiersprache geschrieben wurde, und wandelt dies, indem die Richtigkeit bewahrt wird, in ein effizienteres Programm, ausgedrueckt mit demselben Formalismus, um. Logisches Programmieren kann als eine Erweiterung einer funktionalen Programmiersprache gesehen werden und die Programmtransformationstechniken wurden erfolgreich erweitert um damit umzugehen.

Logisches Programmieren kann als eine signifikante Generalisierung von relationalen Datenbanken angesehen werden. Abfrage-Systeme fuer benutzterorientierte Datenbanken basieren auf Logik. Integritaetsbedingungen, koennen aehnlich wie Programmspezifikationen und -eigenschaften, nur in Formalis-

men ausgedrueckt werden, die die Maechtigkeit symbolischer Logik haben. Also kann logisches Programmieren im einfachsten Fall als elegante Vereinigung von funktionalem Programmieren und relationalen Datenbanken betrachtet werden.

Auch im Bereich der Rechnerarchitektur findet Prolog ein geeignetes Anwendungsgebiet. Forscher haben Programmiersprachen entwickelt, die eine Variable nur einmal mit einem Wert belegen lassen, um den Flaschenhals der durch die uneingeschraenkte Zuweisungsanweisung in konventionellen Programmiersprachen gegeben ist, zu umgehen. (Datenflussrechner)
Diese unkonventionellen Programmiersprachen koennen alls eingeschraenkte Form der funktionalen Programmierung betrachtet werden, was ein Spezialfall der relationalen Programmierung ist, was wiederum ein Spezialfall des logischen Programmierens ist.

Im Gebiet der Kuenstlichen Intelligenz konnten mit Expertensystemen bereits entscheidende kommerzielle Erfolge erzielt werden. Diese Expertensysteme werden mittels Programmiersprachen die auf Ableitungsregeln basieren implementiert, grossteils in LISP. Solche Regeln haben die Form einer Wenn-Dann Anweisung. Wenn B und C, dann A was sehr eng mit der Hornklauseln Teilmenge von Logik verbunden ist. In juengster Zeit beginnt man immer mehr Expertensysteme in Prolog zu implementieren. Expertensysteme die Expertenwissen in einem eingegrenzten Bereich darstellen, sind die be-

sten Beispiele fuer Prolog Applikationen. Der Experte gibt sein Wissen in Prolog in Fakten und Regeln an. die Regeln kontrollieren die Bedeutung der Bezuege die zwischen Fakten hergestellt werden. Prolog versucht wiederholt sukzessiv alle Fakten in der Datenbank zu finden, die die Regeln erfuellen (mittels Backtracking). Falls die Menge der Regeln unvollstaendig ist, dann wird Prolog nicht faehig sein alle Antworten zu geben, die der Experte erwartet, wodurch eine Konsistenz-kontrolle moeglich ist.

Urspruenglich wurde Prolog fuer KI-Zwecke konzipiert. Doch es kann auch als formale Spezifikationssprache fuer Softwaresysteme verwendet werden, da seine unterlagerte Semantik einen sehr hohen Grad der Abstraktion vorsieht.

3. Globale Problemkreise eines Prolog-Uebersetzers

Die Problemschwerpunkte bei der Implementierung eines Prologuebersetzers

liegen im Entwurf folgender Prozesse:

1) rekursive Prozeduraufrufe
2) unifizieren
3) Backtracking
4) cut

3.1. Rekursionen in Prolog

Rekursionen nehmen in Prolog eine zentrale Stellung ein. Sie ersetzen in Prolog auch die aus konventionellen Programmiersprachen bekannten Schleifenformen. So wird zum Beispiel in Prolog folgende Schleife wie folgt geschrieben:

```
while (liste1 != nil) do
   begin
   naechstes_element_von_liste3 :=
            naechstes_element_von_liste1;
   zeiger auf liste1 und auf liste2
            um 1 weiterschalten;
   end;
ende_der_liste3 := liste2;


append(nil,Liste,Liste).
    /* Endbedingung, und dazugehoerige
       abschliessende Aktion */
```

```
append(cons(Element,Liste1),Liste2,cons(Element,Liste3)) :-
   /* erstes Element der Liste1 selektieren
      und damit die Liste3 initialisieren */
   append(Liste1,Liste2,Liste3).
   /* und Schleife weiter ausfuehren */
```

Prolog zeigt hier nicht nur eine elegantere Formulierung der Schleife, auch der Algorithmus wird verstaendlicher. Zudem kann der hier vorgestellte Algorithmus in Prolog nicht nur zwei Listen mit beliebigen Datenstrukturen verketten, sondern kann je nach Art der Parameter auch noch andere Leistungen bieten.

append(Liste1,Liste2,Liste3) liefert:

	Liste1	Liste2	Liste3
a)	Liste	Liste	Liste
b)	Variable	Liste	Liste
c)	Liste	Variable	Liste
d)	Variable	Variable	Liste
e)	Liste	Liste	Variable
f)	Variable	Liste	Variable
g)	Liste	Variable	Variable
h)	Variable	Variable	Variable

a) Ergibt Liste1 verkettet mit Liste2 die Liste3 ?

b) Liefere jene Liste, die verkettet mit Liste2 die Liste3 ergibt

c) Liefere eine Liste, so dass die Liste1 verkettet mit dieser Liste die Liste3 liefert.

d) Liefere alle moeglichen Aufteilungen der Liste3 in zwei Listen.

e) Liefere die Verkettung von Liste1 und Liste2.

f) Liefere alle moeglichen Listen, von denen Liste2 eine Teilliste ist.

g) Liefere alle moeglichen Listen, die mit Liste1 beginnen.

h) Liefere alle moeglichen Listen, und deren moegliche Aufteilungen.

Die Implementierung von Rekursionen ist bereits von Uebersetzern anderer hoher Programmiersprachen bekannt. Sie wird normalerweise mit Hilfe eines oder mehrerer Stacks durchgefuehrt. In unserer Implementierung wird zur Loesung von Rekursionen ein Stack verwendet. Prolog fordert jedoch einen wesentlichen Unterschied im Implementieren dieses Problems gegenueber konventionellen Programmiersprachen. Normalerweise kann der benoetigte Speicherplatz am Stack unmittelbar nach Beendigung eines Prozeduraufrufes wieder freigegeben werden. In Prolog ist dieser einfache Mechanismus (beim Aufruf Speicherplatz an der Spitze des Stack besetzen, und nach Beendigung der Prozedur diesen Speicherplatz wieder

freigeben) jedoch nicht anwendbar, da diese Programmiersprache indeterministische Konstruktionen erlaubt. Dies bedeutet, dass eine Prozedur nach ihrer Beendigung durch Backtracking wieder reaktiviert werden kann, um weitere moegliche Ergebnisse zu liefern.
Die Freigabe des besetzten Speicherplatzes muss daher normalerweise verzoegert werden, bis die Prozedur durch Backtracking ihr letztes Resultat zurueckgeliefert hat.

Es sei noch ein kurzes Beispiel einer indeterministischen Prozedur angefuehrt, um eventuell durch diesen Begriff entstandene Verwirrung zu beseitigen:

```
op(250,und,xfx).

weiblich(anna).
weiblich(maria).
weiblich(eva).
weiblich(barbara).

maennlich(kurt).
maennlich(andreas).
maennlich(hubert).

paar(Frau,Mann) :-
    weiblich(Frau),
    maennlich(Mann),
    write(Frau und Mann),
    fail.
paar(_,_).
```

Ein Aufruf von 'paar' wird als Ausgabe alle 12 moeglichen Paare von Maennern und Frauen liefern. Dies ist nur moeglich, da in 'paar' zwei indeterministische Praedikate (weiblich, maennlich) benuetzt werden, welche durch Backtracking, das durch 'fail' ausgeloest wird, mehrere Resultate liefern koennen.
Nachdem 'weiblich' zum ersten mal mit einer Variable aufgerufen wurde, wird sie 'anna' liefern. Dies ist jedoch nicht das einzig moegliche Resultat dieser Prozedur, sondern sie kann noch drei weiter Resultate durch Backtracking liefern.

3.2. Der Unifizierungsprozess

Der Unifizierungsprozess (Identifizieren von zwei Termen) in Prolog entspricht den Tests und Zuweisungen in konventionellen Programmiersprachen, deren Implementierung kein grosses Problem darstellt. Die eigentliche Schwierigkeit besteht in der Darstellung der neu erzeugten Terme, bzw. der Bindung von Variablen. Zu dieser Problemstellung gibt es zwei bekannte Loesungsverfahren: Structure sharing und pure code copying. In die\ser Implementierung fiel die Entscheidung fuer das Konzept des pure code copying, da sich dieses Konzept wesentlich einfacher implementieren laesst, wodurch auch zur Laufzeit eine bessere Leistung erzielt wird. Zudem wurde von David Warren nachgewiesen (siehe /WA83/), dass structure sharing im Normalfall nur einen zu vernachlaessigenden Vorteil bezueglich Speicherplatzerspannis bringt. (Durch structure sharing benoetigen neu erzeugte Strukturen

im allgemeinen weniger Speicherplatz, dafuer koennen Variable sehr schwer freigegeben werden).

3.2.1. Variable in Prolog

Variable in Prolog koennen zwei Zustaende haben:

undefiniert:

d.h. sie ist noch mit keinem Wert belegt. Dieser Zustand ist nicht mit undefinierten Variablen in konventionellen Programmiersprachen zu vergleichen, wo die Variable vor der ersten Wertzuweisung einen belieben Wert enthaelt, bzw. einen Initialisierungswert, der jedoch als solcher nicht eindeutig erkannt werden kann. 'Undefiniert' hat in diesem Zusammenhang eine Bedeutung, die mit dem Wert 'nil' in PASCAL vergleichbar ist.

Nur undefinierten Variablen kann in Prolog ein Wert zugewiesen werden.

definiert:

Das heisst, die Variable ist an einen Wert gebunden, der eine Konstante (Atom oder Integer), oder eine Struktur sein kann. Eine Wertzuweisung an eine Variable kann nur einmal erfolgen. Werte von Variablen koennen nicht mehr ueberschrieben werden. Durch diese Eigenschaft scheinen Prologprogramme zwar schwierig zu erstellen sein, es zeigt sich jedoch, dass dadurch Programme viel einfacher lesbar, und weniger fehlerhaft

geschrieben werden koennen, bzw. Fehler leicht eruierbar und verbessbar sind.

Werden zwei undefinierte Variable gebunden (unifiziert), so resultiert aus dieser Bindung wieder eine undefinierte Variable. Die Bindung wird durch eine sogenannte Referenz vermerkt. Das heisst, eine der beiden Variablen erhaelt einen Verweis auf die andere Variable zugewiesen. Dadurch koennen durch mehrfaches Binden von verschiedenen Variablen Zeigerketten entstehen, die baumartig organisiert sind, wobei die Wurzel dieses 'Referenzbaumes' den Wert (definiert oder undefiniert) der Variablen beinhaltet. Auf diese Weise werden durch eine einzige Wertzuweisung an eine Variable, alle an diese Variable gebundenen Variablen ebenfalls an diesen Wert gebunden.

Durch die oben genannte Art des Bindens von undefinierten Variablen, wird es noetig eine Variable vor jedem Test, bzw. vor jeder Wertzuweisung vollstaendig zu dereferenzieren. Dereferenzieren bedeutet, dass zu einer Variable das Ende der Referenzliste, also die Wurzel des Referenzbaumes, gesucht wird, da nur der Inhalt dieses Bereiches den Wert einer Variable enthaelt bzw. veraendert werden darf.

3.2.2. Pure code copying

Beim pure code copying wird nicht wie beim structure sharing nur fuer sich aendernde Teile einer Struktur neuer Speicherplatz besetzt, sondern bei der Bindung einer Struk-

tur an eine Variable wird jeweils die gesamte Struktur kopiert. Dadurch wird eine bestimmte Redundanz in Kauf genommen (konstante Teile einer Struktur werden mehrfach kopiert), wobei allerdings der Verweis auf den Quellterm wegfaellt, und auch die Unterscheidung zwischen Quellterm und konstruiertem Term nicht mehr getroffen werden muss. Auf einen eigenen Variablenvektor fuer die Darstellung einer Struktur kann ebenfalls verzichtet werden, da die Variablen nun direkt in den Code fuer eine Struktur integriert sind. Weiters koennen Strukturen in der Weise kodiert werden, dass zuerst sein Funktor, und hierauf sequentiell seine Argumente abgespeichert werden, wodurch ein sehr einfaches und effizientes Verarbeiten von Strukturen ermoeglicht wird.

3.2.3. Unifizieren von zwei Termen (bei code copying)

Vor dem Unifizieren von Termen, muessen diese vollstaendig dereferenziert werden. Hierauf gibt es folgende Fallunterscheidungen:

a) Sind beide Terme Variable, so erhaelt eine Variable einen Verweis auf die andere.

b) Ist ein Term eine Variable und der andere eine Konstante, so wird die Variable mit der Konstanten gebunden, d.h. sie erhaelt die Konstante zugewiesen.

c) Ist ein Term eine Variable und der andere eine Struktur, so erhaelt die Variable einen Verweis auf die Struktur.

d) Sind beide Terme Konstante, so erfolgt bei Gleichheit keine Operation; sonst wird das Backtracking eingeleitet.

e) Sind beide Terme Strukturen, so erfolgt, falls sie verschiedene Funktoren besitzen, oder von verschiedener Stelligkeit sind, das Backtracking; andernfalls werden alle ihre Argumente unifiziert.

f) Ist ein Term eine Konstante und der andere eine Struktur so wird das Backtracking eingeleitet.

3.3. Backtracking

Beim Backtracking muss ein vorangegangener Programmzustand moeglichst schnell wieder hergestellt werden koennen. Hierzu muss die Moeglichkeit vorgesehen werden, dass ein Programmzustand in der Weise aufbewahrt werden kann, dass er aus den daraus resultierenden Informationen jederzeit wieder hergestellt werden kann. Es gibt hierzu bereits aus mehreren experimentellen Sprachen Loesungsvorschlaege. Meistens bauen sie jedoch auf ein bestehendes deterministisches Modell auf. Da Backtracking jedoch ein wesentlicher Bestandteil von Prolog ist, wird es auch weniger vom uebrigen Entwurf des Uebersetzers getrennt werden koennen, ohne erhebliche Leistungseinbussen in der Ausfuehrung des uebersetzten Programmes in Kauf zu nehmen.

Um das Backtracking zu ermoeglichen, wird beim Aufruf einer Prozedur, die aus mehr als einer Klausel besteht, und

daher durch Backtracking oefters ausgefuehrt werden kann, der momentane Systemstatus auf einen Stack gerettet. Die Programmstelle, an der das Retten des Systemstatus noetig ist, wie auch die Datenstruktur, die fuer die Aufnahme der entsprechenden Information benutzt wird, werden choicepoint genannt. Ein choicepoint muss aufbewahrt werden, bis die Prozedure ihr letztes Resultat durch Backtracking zu errechnen beginnt, bzw. bis durch cut der Indeterminismus einer Prozedur aufgehoben wird, und somit der choicepoint aufgeloest werden muss.

Backtracking erfolgt, wenn beim Unifizieren ein Fehler (fail) auftritt, d.h. zwei verschiedene definerte Terme unifiziert werden sollen. Hierauf erfolgt ein Aufruf der Operation 'fail', welche das Backtracking durchfuehrt, indem alle Bindungen, welche zwischen dem choipepoint und dem fail durchgefuehrt wurden rueckgaengig gemacht werden, und der restliche Systemstatus (Zeiger auf die Spitze verschiedener Stacks, wiederherstellen des zum choicepoint gehoerenden Environments und der Uebergabeparameter an die erneut zu berechnende Prozedur) zum Zeitpunkt des choicepoint wieder hergestellt wird. Bindungen koennen dadurch wieder rueckgaengig gemacht werden, indem alle kritischen Bindungen, d.h. solche Bindungen, die im Falle eines fails rueckgaengig gemacht werden muessen, auf einem speziellen Stack (Trail) vermerkt werden.

3.4. Die cut Operation

Die cut-Operation ermoeglicht es eine Prozedur zu determinisieren. Dadurch koennen vom System jene Informationen, die fuer ein Backtracking gespeichert wurden, wieder freigegeben werden. Durch diese Eigenschaft des cut's, wird es auch haeufig benutzt, um das System bei der Speicherfreigabe zu unterstuetzen, bzw. um zu verhindern, dass das System Zeit verbraucht, indem es unnoetige Loesungen sucht. Die cut Operation kann auf drei Arten interpretiert werden:

a) Einmal wird das System nur informiert, dass es dann, wenn es jene Klausel, in der das cut steht, durchlaeuft, auf dem richtigen Weg zur Loesungsfindung ist.

b) Ein cut gefolgt von einem 'fail', informiert das System, dass es weitere Versuche die bearbeitete Prozedure erfolgreich zu beenden unterlassen soll.

c) Die dritte Verwendungsart von cut, gibt dem System bekannt, dass die eben gesuchte Loesung, die einzig moegliche, bzw. sinnvolle ist, und daher weitere Alternativen zur Berechnung einer Prozedur nicht mehr gesucht werden muessen.

Die Implementierung des cut erweist sich als sehr einfach. Es genuegt alle choicepoints, die nach dem aktuellen Environment stehen zu entfernen.
Weiters kann eine garbage collection am Trail durchgefuehrt werden, indem alle Vermerke von Bindungen, die sich nach der

cut Operation als nicht mehr kritisch erweisen aus dem Trail entfernt werden. Diese garbarge collection ist fuer ein fehlerfreies Arbeiten des Systems jedoch nicht unbedingt noetig, da bei einem nachfolgenden fail ohne dieser Massnahme lediglich Bindungen rueckgaengig gemacht wuerden, die durch die Freigabe eines Teils des Stacks nicht mehr zurueckgesetzt werden muessten.

4. Die Prolog-Maschine

4.1. Datenbereiche

Es wird grundsaetzlich zwischen zwei Datenbereichen unterschieden:

- Programmcodebereich
- Datenbereich

Der Programmcodebereich enthaelt die uebersetzten Prologprozeduren, wobei der Programmcode sowohl aus einer zu interpretierenden Zwischensprache, als auch aus Maschinencode bestehen kann.

Die Datenbereiche koennen in vier Kategorien eingeteilt werden:

Der Heap dient zum Speichern von waehrend der Programmausfuehrung neu generierten Strukturen. Auf ihn erfolgt ein wahlweiser Zugriff. Zeiger innerhalb des Heaps muessen keine bestimmte Richtung aufweisen. Der Heap darf keine Verweise in den Stack aufweisen, um inkonsistente Verweise beim Abbau des Stacks zu verhindern. Der Heap waechst bei Prozeduraufrufen und kontrahiert nur beim Backtracking.

Der Stack dient zur Verwaltung von Environments und Choicepoints.

Ein Environment beinhaltet die permanenten Variablen einer Klausel und die Addresse jenes Praedikats, bei dem nach der Ausfuehrung der Klausel die Abarbeitung des Programms fortfahren soll. Zeitlich frueher generierte Environments duerfen keine Zeiger auf zeitlich spaeter errichtete Environments enthalten. (D.h. permanente Variable duerfen nur Verweise auf permanente Variable aus vorangehenden Environments enthalten). Diese Zusicherung ermoeglicht ein einfaches Kontrahieren des Stacks, ohne der Gefahr, inkonsistene Zeigerketten zu erzeugen, ausgeliefert zu sein. Da Environments und Choicepoints im Stack gemischt auftreten koennen, werden die einzelnen Environments zudem noch rueckwaertsverkettet.

Ein Choicepoint enthaelt jene Informationen, die beim Backtracking benoetigt wird, um einen vorangegangen Systemzustand wieder herstellen zu koennen. Er enthaelt die Werte, die folgende Register bei seiner Konstruktion besitzen:

choicepoint pointer

continuation pointer

Environment pointer

Heap pointer

Trail pointer

A-Register (Parameter der Prozedur)

Weiters enthaelt er noch einen Zeiger auf jene Instruktion, bei der bei einem eventuellem Backtracking fort-

gefahren werden soll. (entspricht einem Zeiger auf eine alternative Klausel, und ist nicht mit dem continuation pointer zu verwechseln, der ein Zeiger auf jene Instruktion ist, bei der nach erfolgreichem Ende eines Prozeduraufrufes fortgefahren werden soll)

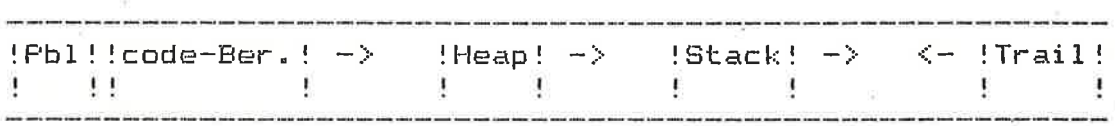
Der Stack waechst beim Aufruf von Prozeduren, er kann jedoch bereits vor dem Erreichen eines Backtrackpunktes wieder schrumpfen. So wird er beim Bearbeiten von cuts (!) verkleinert, ein Environment kann sogar beim Beenden eines Prozeduraufrufes freigegeben werden, falls dem Environment kein choicepoint folgt, indem alle Informationen, die noch im Environment liegen auf den Heap gerettet werden. Durch dieses Verfahren wird auch in einfacher Weise eine Speicherplatzoptimierte Abarbeitung von tail-Rekursionen ermoeglicht. (d.h. Rechtsrekursionen werden durch Schleifen ersetzt).

Der Trail enthaelt Informationen, um beim Backtracking, Variable wieder zuruecksetzen zu koennen (auf 'undefiniert' setzen), welche nach dem choicepoint gebunden wurden. Er waechst beim Binden von Variablen, und kontrahiert beim Backtracking.

Der Parameterblock dient zur Uebergabe von Parametern an eine Prozedur. Er wird vor einem Prozeduraufruf initialisiert und nach dem Abarbeiten des Head wieder freigegeben. Um beim Backtracking seinen Zustand wieder herstellen zu koennen, wird sein Inhalt, falls noetig, in einem choicepoint aufbewahrt. Ausserdem dient der Parameterblock als

Speicher fuer temporaere Variable.

Der vorhandene Speicherbereich wird einem Prologprogramm wie folgt zugeteilt:



4.2. Spezielle Register:

P (program pointer) dieses Register dient als Zeiger auf den auszufuehrenden Programmcode

CP (continuation pointer) dieses Register dient als Zeiger auf jene Anweisung, bei der, nach dem Abarbeiten des Codes einer Klausel, fortgefahren werden soll. Vereinfacht kann man sagen, dass der continuation pointer die Ruecksprungaddresse einer Klausel enthaelt.

E (Environment pointer) dieses Register zeigt auf das Environment der gerade aktiven Klausel.

B (choicepoint pointer) dieses Register zeigt auf den zuletzt erzeugten choicepoint. Es ist also ein Zeiger auf jenen choicepoint, bei dem nach einem fail die Programmausfuehrung fortfahren soll.

TR (Trail pointer) dieses Register zeigt die Spitze des Trails an.

H (Heap pointer) dieses Register zeigt die Spitze des Heap an.

S (structure pointer). Dies ist ein Zeiger auf eine am Heap liegende Prolog-Struktur, deren Parameter mittels diesem Zeiger sequentiell unifiziert werden.

A-Register

(Argument Register). Sie werden so weit als moeglich in Registern der CPU aufbewahrt. Sie enthalten die Parameter, welche einer aufgerufenen Prozedur uebergeben werden. Ihr Standort befindet sich im Parameterblock (sofern sie in Hardware-registern keinen Platz mehr finden).

Y-Register

(Register fuer permanente Variable). Sie stehen in einem Environment. Sie duerfen nach Beendigung eines Prozeduraufrufes nicht freigegeben werden, (da sie gegebenenfalls fuer ein Backtracking noch benoetigt werden), es sei denn, sie verweisen nicht in das eigene Environment.

X-Register

(Register fuer temporaere Variable). Sie stehen im Parameterblock. Xi und Ai benutzen denselben Speicherplatz. Da sie nur in einem Praedikat einer Klausel

auftreten, koennen sie sofort nach ihrem Gebrauch (d.h. nach Beendigung der Prozedur, der sie uebergeben werden) wieder freigegeben werden.

pbl (parameterblock). Zeiger auf den Parameterblock. Dies ist eine Konstante und entspricht der Addresse des Registers A1.

V-Register

Steht anstelle Yi, bzw Xi.

4.3. Variablentypen

Variablen werden, ihrer Lebensdauer entsprechend, in drei Kategorien eingeteilt:

Temporaere Variable:

Ihre Lebensdauer ist auf die Verarbeitung eines Praedikats beschraenkt. In diese Klasse fallen alle Variable, welche nur in einem Praedikat benoetigt werden (wobei der Kopf einer Klausel zum ersten Praedikat zu zaehlen ist). (David Warren schraenkte temporaere Variable staerker ein, indem er zudem noch forderte dass sie ihr erstes Auftreten im Kopf einer Klausel, in einer Struktur, bzw. im letzten Praedikat haben. Diese Einschraenkung kann jedoch zu einem inkonsistenen Systemzustand fuehren, da so ein Verweis auf eine Variable im Environment uebergeben werden kann (put_pVa-

riable Yi, Aj) die bereits freigegeben wurde (naemlich dann, wenn die Variable nur in einem einzigen Goal auftritt)).

Permanente Variable:

Alle Variable die nicht die Bedingung einer temporaeren Variable erfuellen gelten als permanent. Permanente Variable werden nach ihrem letzten Auftreten in der Klausel am Heap noetigenfalls gesichert, und hierauf im Environment freigegeben.

Void Variable:

Diese Variablen haben die Eigenschaft, dass sie nur einmal in einer Klausel auftreten. Sie benoetigen keinen Speicherplatz. Falls sie im Kopf einer Klausel auftreten muessen sie nicht bearbeitet werden.

4.4. Der Befehlssatz

Die Prologmaschine kennt sechs Arten von Instruktionstypen:

- get, unifizieren von Argumenten im Kopf einer Klausel mit den uebergebenen Parametern.
- put, Uebergabe von Parametern an eine Prozedur
- unify, unifizieren von Argumenten einer Struktur im Head.

- match, unifizieren von Argumenten einer Struktur im body.

- Instruktionen fuer den Kontrollablauf

- Backtrack- und Hash-Instruktionen

GET-Instruktionen entsprechen den Argumenten des Head einer Klausel, und muessen die Parameter im Head (Yi, Xi) mit den uebergebenen Parametern (Ai) unifizieren, und gegebenenfalls eine Fehlerbehandlung einleiten (Backtracking).

PUT-Instruktionen entstprechen den zu uebergebenden Parametern. Sie muessen auch Sorge tragen, dass Variable initialisiert werden, bzw. 'unsichere Variable' (Permanente Variable, die nach dem Aufruf eines Praedikats freigegeben werden, obwohl sie einen Verweis auf das momentane Environment beinhalten, wodurch es zu ungueltigen Zeigerverweisen kommen kann) auf dem Heap gesichert werden.

UNIFY-Intruktionen entsprechen den Argumenten von Strukturen. Bei unify-Instruktionen muss darauf geachtet werden, dass im Heap keine Verweise auf Environments erzeugt werden, wodurch der Heap, beim Kontrahieren des Stacks, inkonsisten Verweise enthalten wuerde. unify-Instruktionen sind in ihrer Wirkungsweise von einem Zustand (Lesen (analysieren) - Schreiben (aufbauen)) abhaengig. Beim Schreiben werden auf dem Heap neue Strukturen erzeugt. Beim Lesen,

wird versucht eine bestehende Struktur mit der gegebenen Struktur zu unifizieren. Da die Parameter einer Struktur, dem Strukturnamen unmittelbar und sequentiell folgen, ist eine besondere Behandlung von geschachtelten Strukturen noetig. Eine geschachtelte Struktur wird stets wie eine Variable behandelt welche an die geschachtelte Struktur gebunden ist. Hierbei werden geschachtelte Strukturen von der aeussersten Schachtelungsebene ausgehend bis zur innersten Schachtelungsebene analysiert.

MATCH-Instruktionen entsprechen den Argumenten von Strukturen im body einer Klausel. Im Prinzip entsprechen diese Instruktionen den 'unify'-Instruktionen im write modus. Geschachtelte Strukturen werden jedoch nicht wie Variable behandelt an denen die Struktur gebunden werden, sondern sie werden mittels eines relativen Offsets direkt gebunden. (D.h. zuerst wird eine Referenz auf eine Struktur erzeugt, welche erst zu einem spaeteren Zeitpunkt erzeugt wird. Hieraus folgt natuerlich eine temporaere Inkonsistenz des Heap).

Kontroll-Instruktionen dienen zur prozeduralen Ablaufsteuerung von Prologprogrammen. (Speicherplatzverwaltung, Uebergabe der Ablaufkontrolle an neue Klauseln, Behandlung von Fehlern, Backtracking, cut).

Indizierungs-Intruktionen bilden eine Erweiterung des Prologuebersetzers, welche fuer einen korrekten Programmablauf nicht noetig waeren, jedoch eine gezielte Indizierung

von geeigneten Klauseln erlaubt, wodurch der Programmablauf beschleunigt werden kann. Die Auswahl einer geeigneten Klausel erfolgt durch Analyse des ersten Parameters. Sollte es sich hierbei um eine Konstante bzw. eine Struktur handeln, so wird hierbei mittels eines hash-Verfahrens nach Klauseln gesucht, welche ein erfolgreiches Unifizieren des ersten Parameters garantieren.

4.4.1. Beschreibung der einzelnen Instruktionen

4.4.1.1. Kontroll-Instruktionen

ALLOCATE

Diese Instruktion leitet eine Klausel mit mehr als einem Praedikat im body ein. Sie bewirkt, dass der Klausel Platz fuer ihr Environment bereitgestellt wird. Das Environment wird am top-of-Stack erstellt, d.h. es kann unmittelbar einem Environment bzw. einem choicepoint folgen, je nachdem was am Stackende stand. Im Environemnt werden die permanenten Variablen der Klausel aufbewahrt, und ein Zeiger auf das vorangehende Environment sowie der continuation-pointer gespeichert.

```
ce = e;
e = (e < b) ? b : e + env_size(cp);
/* env_size(cp) wird als parameter von 'call' zur
   Verfuegung gestellt  */
e(off_cep) = cp;
e(off_ce) = ce;
```

DEALLCOCATE

Diese Instruktion geht der execute Instruktion einer Klausel unmittelbar voran. Durch sie wird das Environment der Klausel freigegeben und der vorangegangene continuation pointer wieder hergestellt. Auswirkungen auf den Speicher platz hat diese Instruktion jedoch nur, falls waehrend der Ausfuehrung der Klausel kein

choicepoint erzeugt wurde.

```
cp = e(off_cep);
e = e(off_ce);
```

CALL Proc,N

Diese Intruktion wird nach dem Bereitstellen der Uebergabeparameter fuer die Prozedur 'Proc' ausgefuehrt. Die Ablaufkontrolle wird der Prozedur 'Proc' uebergeben, nachdem der continuation pointer auf das nachfolgende Praedikat gesetzt wurde. N gibt die Anzahl der zur Zeit im Environment befindlichen Variablen an.

```
cp = next_code;
p = Proc;
```

EXECUTE Proc

Diese Instruktion ersetzt beim letzten Praedikat einer Klausel die Anweisung 'call'. Zum Unterschied von call wird jedoch der continuation pointer unveraendert belassen, wodurch nach dem Aufruf des Praedikats die Ablaufkontrolle jenem Praedikat ueberlassen wird, welches die gerade aktive Prozedur aufgerufen hat.

```
p = Proc;
```

PROCEED

Diese Intstruktion schliesst eine 'unit Klausel' (eine Klausel ohne body) ab. Sie uebergibt die Ablaufkontrol-

le jener Klausel auf die der continuation pointer verweist.

```
p = cp;
```

FAIL

Diese Instruktion wird beim Auftreten eines Fehlers (zwei Argumente koennen nicht unifiziert werden) aufgerufen. Sie veranlasst, dass die Programmausfuehrung am letzten choicepoint wieder aufgenommen wird, nachdem der Systemstatus, mittels der im choicepoint enthaltenen Informationen, wieder auf den zum Zeitpunkt der Choicepointerstellung gueltigen Stand gebracht wurde, und in der Zwischenzeit erfolgte Bindungen, mittels der Informationen im Trail, rueckgaengig gemacht wurden.

```
unwoud(tr);
reset_register(b);
```

TRAIL R

wird aufgerufen falls eine Variable mit der Referenz R waehrend einer Unifizierung gebunden wird. Um eine solche Bindung beim Backtracking rueckgaengig machen zu koennen wird R auf dem Trail vermerkt falls gilt:

R liegt im Heap vor dem zum letzen Choicepoint gehoerigen Heap pointer

oder R liegt im Environment Stack nicht im gerade aktiven Environment.

CUT

bewirkt, dass die gerade aktive Klausel deterministiert wird, d.h. alle choicepoints, die nach dem Aufruf der gerade aktiven Klausel angelegt wurden, werden vom choicepoint-Stack entfernt. Jene Referenzen am Trail, welche die Bedingung um dort aufgenommen zu werden nun nicht mehr erfuellen, werden dort entfernt. (am Heap koennte eine garbage collection stattfinden).

TRY_ME_ELSE L

Diese Instruktion geht dem Code der ersten Klausel einer Prozedur voran, die aus mehr als einer Klausel besteht. Ein choicepoint wird am top-of-Stack erstellt, wobei L als naechste offene (d.h. alternative) Klausel vermerkt wird. Der Heap pointer, der Environment pointer, der Trailpointer und die uebergebenen Parameter werden gesichert.

RETRY_ME_ELSE L

Diese Instruktion steht vor einer Klausel, die weder die erste noch die letzte innerhalb einer Prozedur ist. Im von try_me_else erzeugten choicepoint wird lediglich die naechste alternative Klausel auf L gesetzt, sowie die Parameter fuer die Prozedure wieder richtig gesetzt.

```
b(off_p) = L;
copy_args(arg_count,&b(off_a1),pbl);
```

TRUST_ME_ELSE fail

Diese Instruktion gibt den durch try_me_else erzeugten choicepoint wieder frei.

```
reset_parameter;
b = b(off_b);
```

TRY L

Erste Instruktion zum Aufruf einer Folge von Klauseln mit demselben Schluessel als erstes Argument. Ein choicepoint wird am top-of-Stack erstellt, wobei der zeiger auf eine alternative Klausel auf die naechste Instruktion nach try 1 gesetzt wird. Der Heap pointer, der Environment pointer, der Trailpointer und die uebergebenen Parameter werden gesichert. Die Ablaufkontrolle wird sodann L uebergeben

RETRY L

Steht in der Mitte einer Liste von Aufrufen von Klauseln mit demselben Schluessel als erstem Element. Der von try erstellte choicepoint wird veraendert, indem die alternative Klausel auf die retry 1 folgende Instruktion gesetzt wird. Die Ablaufkontrolle wird L uebergeben.

```
b(off_p) = next_code;
reset_parameter;
p = L;
```

TRUST L

Steht als letzte Instruktion in einer Reihe von Aufrufen von Klauseln mit demselben Schluessel als erstem Argument. Der von try erzeugte choicepoint wird freigegeben, und hierauf die Ablaufkontrolle L uebergeben.

```
reset_parameter;
b = b(off_b);
p = L;
```

4.4.1.2. PUT-Intstruktionen

PUT_VARIABLE Yn,Ai

steht fuer einem Praedikat-argument, welces eine permanente Variable ist, die zum ersten mal benutzt wird, (d.h. sie ist sicher noch undefiniert). Yn wird auf undefiniert gesetzt (zeigt auf sich selbst) und Ai erhaelt einen Verweis auf Yn.

```
Ai = Yn = env_ref(n);
```

PUT_VARIALBE Xn,Ai

steht fuer einem Praedikat-argument, welches eine temporaere Variable ist, die zum ersten mal benutzt wird, und daher noch undefiniert sein muss. Hierbei wird der Heap um eine ungebundene Variable erweitert, auf welche sowohl Xn als auch Ai verweisen.

```
Ai = Xn = Heap_ref;
```

PUT_VOID Ai

steht fuer einem Praedikat-argument, welches eine void Variable ist. Hierbei muss am Heap eine ungebundene Variable erzeugt werden, und Ai eine Referenz auf diese Variable zugewiesen werden.

Ai = Heap_ref;

PUT_VALUE Vn,Ai

steht fuer ein gaol-argument, welches eine Variable ist die bereits gebunden wurde (nach dem ersten verwenden dieser Variable). Hierbei wird Vn einfach nach Ai kopiert.

Ai = Vn;

PUT_UNSAFE_VALUE Yn,Ai

steht im letzten Praedikat in dem eine 'unsichere' Variable auftritt, statt 'put_value'. Yn ist 'unsicher', falls Yn mit 'put_variable' initialisiert wurde. Ist Yn bereits 'sicher' (kein Verweis in das augenblickliche Environment), so erhaelt Ai den dereferenzierten Wert von Yn zugewiesen. Andernfalls wird Yn gesichert, indem auf dem Heap eine ungebundene Variable angelegt wird, auf welche sowohl Yn als auch Ai verweisen. (Diese neue Bindung wird falls noetig am Trail vermerkt).

PUT_CONST C,Ai

stellt ein Praedikat-Argument dar, welches die Konstante C ist. Ai wird C zugewiesen.

```
Ai = C;
```

PUT_STRUCTURE F,Ai

stellt ein Praedikat-Argument dar, welches eine Struktur ist. F wird auf den Heap gegeben, und Ai erhaelt hierauf einen Verweis.

```
Ai = struct_ref(F);
```

4.4.1.3. GET-Instruktionen

GET_VARIABLE Vn,Ai

Vn ist ein Head Argument, das zum ersten mal unifiziert wird, d.h. Vn ist sicher noch undefiniert. Also genuegt, dass Ai nach Vn kopiert wird.

```
Vn = Ai;
```

GET_VOID Ai

Eine Void Variable soll unifiziert werden. Diese Instruktion entspricht einem 'nop'.

```
;
```

GET_VALUE Vn,Ai

Vn ist ein Head argument, aber bereits gebunden (d.h.

wurde bereits mit get-Variable gebunden). Die dereferenzierten Werte von Vn und Ai werden unifiziert. Ist Vn eine temporaere Variable, so darf Vn der voll dereferenziert Ergebniswert zugewiesen werden (daraus folgt eine Verkuerzung von Verweisketten).

GET_CONSTANT C,Ai

Der dereferenzierte Wert von Ai wird mit C unifiziert. Ist Ai eine Variable so erhaelt sie C zugewiesen. Ist Ai initialisiert so muss ihr Wert C entsprechen; andernfalls wird das Backtracking eingeleitet.

GET_STRUCTURE F,Ai

Bewirkt die Behandlung einer Struktur als Heap Parameter. Ai wird dereferenziert. Ist Ai eine Variable, so wird am Heap die Erzeugung einer neuen Struktur eingeleitet. F wird auf den Heap gegeben, und Ai erhaelt einen Verweis hierauf, (der falls noetig am Trail aufbewahrt wird). Die Behandlung der Argumente der Struktur erfolgt im write-Modus. (d.h. unify-instruktionen erzeugen eine Struktur am Heap). Ist Ai eine Struktur und entspricht ihr Funktor F, so wird eine Verweis von S auf das erste Argument von F gegeben. Die weitere Verarbeitung der Strukturelemente erfolgt im read-Modus. (Eine Struktur wird zerstueckelt). Sollte keine von beiden Bedingungen erfuellt sein, so wird eine Fehlerbehandlung eingeleitet.

4.4.1.4. UNIFY-Instruktionen

UNIFY_VOID N

Entspricht einer Reihe von N Variablen die nur 1-mal auftreten. Im read-Modus werden einfach N Parameter uebersprungen. Im write-Modus werden N Parameter auf 'ungebunden' gesetzt.

UNIFY_VARIABLE Vn

Entspricht einer ungebundenen Variable in einer Struktur. Im read-Modus erhaelt Vn den Wert der Variable auf die der structure-pointer S zeigt zugewiesen. Im write-Modus wird eine neue ungebundene Variable am Heap angelegt, auf welche Vn einen Verweis erhaelt.

UNIFY_VALUE Yn

Entspricht einer gebundenen Variable in einer Struktur. Im read-Modus werden der dereferenzierte Wert von Vn und S unifiziert. Das darausfolgende Ergebnis wird Vn zugewiesen, falls Vn eine temporaere Variable ist. Im write-Modus wird auf dem Heap eine neue Variable angelegt, die den Wert von V zugewiesen erhaelt.

UNIFY_LOCAL_VALUE Vn

Entspricht einer gebundenen Variable in einer Struktur, die moeglicherweise einen Verweis in den Environment-Stack enthaelt. Im read-Modus, bzw. falls der derefe-renzierte Wert von Vn nicht ein Verweis in den Environ-

ment-Stack ist entspricht unify_local_value unify_value. Yn ist nicht lokal, falls sie mit unify_variable initiallisiert wurde, bzw. bereits ein 'unify_local_value Vn' voranging.

ist Vn lokal dann gilt:

Auf dem Heap wird eine neue ungebundene Variable angelegt. Vn wird an diese Variable gebunden (Bindung kommt falls noetig auf den Trail), ist Vn eine temporaere Variable so erhaelt Vn direkt einen Verweis auf die neue Variable am Heap.

UNIFY_CONSTANT C

Entspricht einer Konstanten in einer Struktur. Im read-Modus wird der dereferenzierte Wert von S mit C verglichen. Bei Ungleichheit wird eine Fehlerbehandlung eingeleitet. Im write-Modus wird am Heap eine neue Variable angelegt, der C zugewiesen wird.

UNIFY_LAST_STRUCTURE F

Entspricht einer geschachtelten Struktur, die das letzte Argument einer Struktur darstellt. Diese Instruktion ersetzt folgendes Instruktionspaar, wobei das Register Xi nicht benoetigt wird:

unify_variable Xi
get_structure F, Xi

4.4.1.5. MATCH-Instruktionen

MATCH_VOID N

Entspricht einer Reihe von N Variablen die nur 1-mal auftreten. Es werden N Parameter auf 'ungebunden' gesetzt.

MATCH_VARIABLE Vn

Entspricht einer ungebundenen Variable in einer Struktur. Eine neue ungebundene Variable wird am Heap angelegt, auf welche Vn einen Verweis erhaelt.

Vn = Heap_ref;

MATCH_VALUE Vn

Entspricht einer gebundenen Variable in einer Struktur. Auf dem Heap wird eine neue Variable angelegt, die den Wert von V zugewiesen erhaelt.

MATCH_LOCAL_VALUE Vn

Entspricht einer gebundenen Variable in einer Struktur, die moeglicherweise einen Verweis in den Environment-Stack enthaelt. Da keine Verweise vom Heap in den Environment-Stack existieren sollen, muss sichergestellt werden, dass die Variable global ist. Falls der dereferenzierte Wert von Vn nicht ein Verweis in den EnvironmentStack ist entspricht unify_local_value unify_value. Yn ist nicht lokal, falls sie mit unify_variable

initialisiert wurde, bzw. bereits ein 'unify_local_value Yn' voranging. ist Yn lokal dann gilt:

Auf dem Heap wird eine neue ungebundene Variable angelegt. Vn wird an diese Variable gebunden (Bindung kommt falls noetig auf den Trail), ist Vn eine temporaere Variable so erhaelt Vn direkt einen Verweis auf die neue Variable am Heap.

MATCH_CONSTANT C

Entspricht einer Konstanten in einer Struktur. Am Heap wird eine neue Variable angelegt, der C zugewiesen wird.

MATCH_REFERENZ N

Entspricht einer geschachtelten Struktur. Da die Argumente einer Struktur sequentiell aufeinanderfolgend am Head stehen, wird zuerst ein Zeiger auf eine Struktur erzeugt, der dann mittels MATCH_STRUCTURE initialisiert wird. N gibt dabei den relativen Offset der Struktur am Heap zum Heap pointer an.

MATCH_STRUCTURE F

Entspricht einer geschachtelten Struktur. Sie wird am Heap erzeugt. Ein Verweis auf sie, wird durch MATCH_REFERENZ erzeugt.

MATCH_LAST_STRUCTURE F

Entspricht dem letzten Argument einer Struktur, wobei dieses wiederum eine Struktur ist.

match_referenz O

match_structure F

4.4.1.6. Indizierungs-Instruktionen

SWITCH_ON_TYPE Lv,Lc,Ls

Der erste uebergebene Parameter wird ausgewertet und entsprechend seiner Wertklasse wird nach Lv,Lc bzw. Ls verzweigt.

A1 = ungebundene Variable --> goto Lv

A1 = Konstante --> goto Lc

A1 = Struktur --> goto Ls

```
switch (typ(deref(A(1))))
   {case var:  p = Lv;
    case atom: p = Lc;
    case struct: p = Ls;
   }
```

SWITCH_ON_CONSTANT Table

Erlaubt einer Hashtabelle Zugriff auf Klauseln mit einer Konstanten als erstem Argument.

```
p = hash(deref(a(1)),Table,N);
```

SWITCH_ON_STRUCTURE Table

Entspricht switch_on_constant, nur dass diesmal Strukturen als Selektionsmerkmal dienen.

```
p = hash(deref(a(1)),Table,N);
```

5. Alternative Entwurfsvorschlaege

5.1. Structure sharing

Structure sharing bietet eine Moeglichkeit einen Term des Quellprogramms darzustellen. Hierbei wird der urspruengliche Term im Quellprogramm Quellterm und den waehrend der Programmausfuehrung neu konstruierten Term 'konstruierten Term' nennen.
Ein konstruierter Term wird durch ein Zeigerpaar dargestellt.

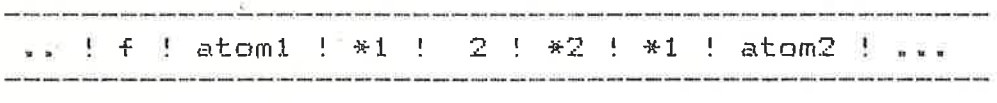
< *Quellterm, *Variablenvektor >

Das Zeigerpaar besteht also aus einem Verweis zum Quellterm und einem Verweis auf einen Variablenvektor, der die Werte der Variablen im Quellterm enthaelt. Der Quellterm stellt also den unveraenderlichen Teil eines Terms dar, eine Art Geruest, und der Variablenvektor jenen Teil des Terms der von Verwendung zu Verwendung variieren kann. Jeder Variable im Quellterm wird eine Nummer zugeordnet, die als Index im Variablenvektor dient, und damit einen schnellen Zugriff auf die Variable erlaubt.
Hierzu ein kurzes Beispiel:

Das Quellfile enthalte die Struktur

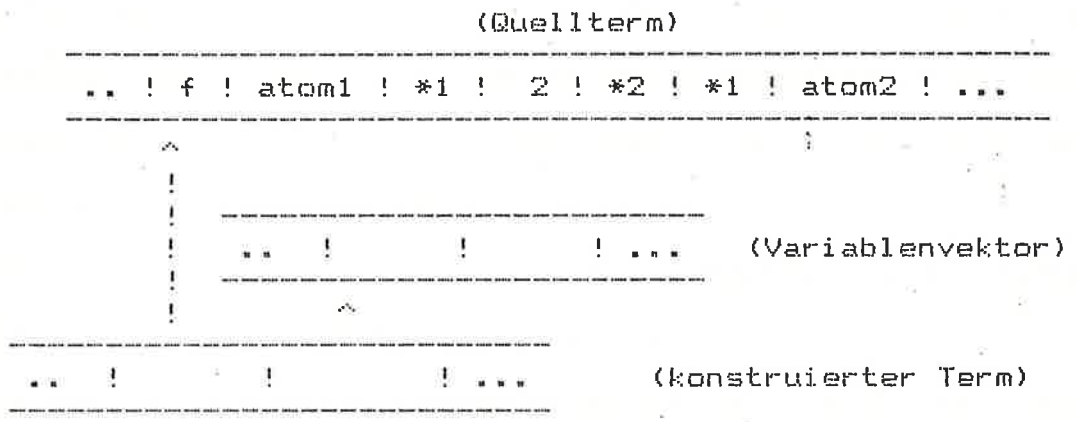
```
f(atom1,Var_i,2,Var_j,Var_i,atom2).
```

Diese Struktur wird als Quellterm wie folgt dargestellt:



/* *i bedeutet Variable mit Nummer i */

Wird diese Struktur an eine Variable gebunden, so erhaelt die Variable einen Verweis auf den Quellterm und zusaetzlich einen Verweis auf einen Variablenvektor.



5.2. Environment-stacking gegen Praedikat-stacking

Der erlaeuterte Entwurf einer Prologmaschine entspricht einem Environment-stacking Modell, wobei, soweit es Terme betrifft, structure sharing nicht verwendet wird. In der Darstellung von Praedikaten wird jedoch weiterhin structure sharing verwendet. (Ein Praedikat wird durch zwei Verweise dargestellt, einem Verweis in den Code fuer das Praedikat und einem auf sein Environment).

Das Praedikat-stacking Modell unterscheidet sich darin vom Environment-stacking Modell, dass es ueberhaupt kein structure sharing verwendet. Nicht nur Terme werden explizit dargestellt, sondern auch Praedikate. Der Stack ent-

haelt eine explizite Darstellung der Liste jener Praedikate, die noch ausgefuehrt werden muessen. (Diese Liste entspricht der 'Schluss' der traditionellen Theorie des logischen Beweisens). Es werden keine Environments mehr benoetigt.

Die Vorteile des Praedikat-stacking Modells liegen in dessen einfachen Implementierung.
Die Tail Rekursion-Optimierung ist wesentlich einfacher zu gestalten und ist auf jede Prozedur anwendbar. Man gibt einfach das aufrufende Praedikat frei, falls es nach dem letztem choicepoint steht.
Alle Variable in einer Klausel sind temporaer und koennen direkt Hardwareregistern zugeordnet werden.
Wenn die Loesung einer Klausel beendet ist, gibt es keine Verweise zum Code dieser Klausel. Diese Tatsache duerfte das Paging in einem virtuellen System sehr reduzieren. Es gibt auch keine Spruenge in einer Klausel mehr. Daraus ergibt sich bei einer Pipeline-Hardware eine schnellere Verarbeitung.

All diese Vorteile koennen jedoch die Nachteile des Modells nicht aufwiegen.
Zeit kann durch unnoetiges Kopieren verbraucht werden; speziell wenn eine Klausel ausgefuehrt wird, die in ihrem Rumpf ein 'fail' erzeugt. Dieser Nachteil ist jedoch nicht so gravierend, da Kopieren relativ schnell (im Vergleich zu anderem Overhead) durchgefuehrt werden kann.
Die Stackgroesse bleibt weniger stabil, wodurch es wenig

Nutzen bringt, aus Optimierungsgruenden die Stackspitze in Hardwareregistern bzw. in einem schnellen Speicher zu halten.
Jedes Praedikat wird vom Stack wieder entfernt, wodurch man unsichere Variable auf den Heap retten muss, um inkonsistente Verweise zu verhindern. Fuer dieses Problem scheint es, nach David Warren, keine elegante Loesung zu geben.
Es ist schwierig, den Code des Rumpfes einer Klausel zu optimieren, da das Preadikat auf den Stack kopiert wird.

5.3. Vor- und Nachteile der Trennung von Environments und Choicepoints.

Im gegenwaertigen Modell muss das aktuelle Environment nicht notwendigerweise an der Spitze des Stacks stehen. Es kann von nachfolgenden Choicepoints ueberlappt worden sein. Indem das Environment in seiner urspruenglichen Position belassen wird, wird Speicherplatz und Kopieraufwand erspart. Allerdings wuerde dies eine Reihe von Vorteilen mit sich bringen.
Permanente und temporaere Variable koennten gemeinsam als Offset von der Spitze des Stacks angesprochen werden.
Environments koennten modifiziert und getrimmt werden, wodurch sie kleiner werden koennten.
Wenn eine Variable dereferenziert wird, ist es moeglich sie zu veraendern, d.h. permanente Variable koennen wie temporaere Variable behandelt werden.
Der Speicherzugriff wuerde nicht mehr so gestreut erfolgen,

was eine bessere Leistung beim paging bzw. beim Zwischenspeichern der Stackspitze ergeben wuerde.
Fuer eine Softwareimplementierung scheinen diese Vorteile jedoch nicht den Kopieraufwand aufzuwiegen.

6. Der Prolog-Uebersetzer

6.1. Projektumgebung und -entwurf

Beim Entwurf des Prolog-Uebersetzers waren folgende Rahmenbedingungen vorgesehen:

b) Der Uebersetzer sollte auf dem Rechnersystem CADMUS 9200 (Fa. PCS) entwickelt werden. Dieses System arbeitet auf der Basis eines MC68010 Prozessors, ist als multiuser-System konzipiert und benuetzt als Betriebssystem MUNIX, (eine fuer diesen Rechnertyp konzipierte UNIX-Version).

b) IF/Prolog, ein Prologinterpreter, der sich durch eine sehr saubere und benutzerfreundliche Schnittstelle fuer in 'C' geschriebene Erweiterungen, und einem sehr leistungsfaehigen debug-Mechanismus auszeichnet, steht fuer den oben genannten Rechner bereits zur Verfuegung.

c) Der Uebersetzer soll eng mit IF/Prolog zusammenarbeiten. Er wird wie eine Art builtin Praedikat verwendet werden. Uebersetzte Prozeduren sollen vom Interpreter aus aufgerufen werden koennen. Weiters sollen einzelne Prozeduren uebersetzt werden koennen und nicht nur vollstaendige Programme.

d) Auf degub-Moeglichkeiten des uebersetzten Codes wird verzichtet, da hierfuer der Interpreter besser geeignet scheint.

e) Der Uebersetzer soll selbst, soweit dies moeglich ist, in Prolog geschrieben werden.

f) Der Uebersetzer uebersetzt den Quellcode in eine assemblerartige Zwischensprache, die den Maschinencode einer Prologmaschine darstellen koennte, wobei fuer die Prologmaschine die Einschraenkung besteht, dass sie speziell durch den Prozessor MC68000 leicht zu emulieren sein soll.

g) In einer ersten Implementierungsstufe wird der erzeugte Zwischencode auf seine Richtigkeit ueberprueft.

h) In einem hierauf folgenden Schritt wird der Zwischencode fuer eine Interpretation durch eine Prologmaschine aufbereitet, wobei die Prologmaschine durch einen in C geschriebenen Interpreter, der in IF/Prolog integriert wird, emuliert werden soll.

i) In einer weiteren Stufe wird der Zwischencode assembliert, und dabei der Maschinencode verschiedenster bestehender Mikroprozessoren erzeugt, wobei speziell Assembler fuer die Prozessoren MC68000 und Intel 8086 geschrieben werden sollen.

j) Abschliessend wird der Uebersetzer um die Gleitkommaverarbeitung erweitert, und mit speziellen Laufzeitoptimierungs-Mechanismen ergaenzt (garbare collection, intelligentes Backtracking, mode Deklarationen, optimierte Hashzugriffe auf Klauseln)

Um diesen Aufgabenbereich abzudecken wurden zwei Projekte parallel gestartet. Eines sollte den bestehenden Interpreter neu bearbeiten, und dessen Datenstrukturen an die der Prologmaschine anpassen, damit ein Zusammenwirken zwischen dieser Maschine und IF/Prolog ermoeglicht wird. Durch dieses Konzept muessen Programme nur teilweise uebersetzt werden, wodurch beim Arbeiten mit dem System sowohl die Vorteile eines Interpreters als auch die Vorteile eines Uebersetzers ausgenuetzt werden koennen. Das zweite Projekt wurde mit dem Entwurf des PrologUebersetzers beauftragt.

Der Prologuebersetzer wurde in zwei grossen Stufen implementiert. Eine erste Implementierung des Uebersetzers sollte dazu dienen, Erfahrungen im Umgang mit Prolog zu sammeln, und alle noetigen Voraussetzungen fuer die Konzipierung eines, fuer die bestehende Hardware optimierten, Zwischencodes zu erlangen. Hierauf sollte der erste Entwurf auf Konsistenz und Schwaechen ueberprueft, und ein neuer, optimierter Uebersetzer entwickelt werden, wobei speziell der Zwischencode optimiert und verfeinert werden sollte.

6.2. Uebersetzerschritte

Ein Prologuebersetzer, der als Praedikat eines Prologinterpreters implementiert ist, hat nur wenige Aufgaben zu erfuellen.

a) Sammeln aller Klauseln einer Prozedur

b) Uebersetzen der einzelnen Klauseln

b1) Die Klauseln muessen auf eine richtige statische Semantik ueberprueft werden.

b2) Geschachtelte Strukturen sind aufzuloesen.

b3) Die Variablen sind in zwei Kategorien aufzugliedern:
temporaer und permanent.

b4) Den Variablen muss ein Speicherplatz zugeordnet werden.

c) Die einzelnen uebersetzten Klauseln muessen mit Instruktionen fuer Backtracking und Hash-Mechanismen verbunden werden.

d) Der uebersetzte Zwischencode wird assembliert.

6.3. Die Schnittstelle zum Prologinterpreter

Da der Uebersetzer selbst ein Prologprogramm ist, und vom Interpreter verarbeitet wird, koennen einige Vorteile von Prolog ausgenuetzt werden. So muss eine zu uebersetzende Prologprozedur nicht als Charakterfile eingelesen werden, sondern sie wird mit Hilfe eines builtin Praedikats von Prolog in aufbereiteter Form direkt zur Verfuegung gestellt.

```
functor(Kopf,Funktor,Stelligkeit).
findall(klausel(Kopf,Rumpf),clause(Kopf,Rumpf),Proze-
dur).
```

liefert in einer Liste alle Prologklauseln, deren Kopf den Funktor 'Funktor' mit der Stelligkeit 'Stelligkeit' besitzen. Hierbei wird ein leerer Rumpf einer Klausel durch 'true' dargestellt. Durch diese beiden Anweisungen erueberigt sich sowohl das Parsen als auch die Syntaxanalyse einer Prologprozedur. Prolog liefert naemlich eine Liste die den Kopf und den Rumpf aller Klauseln jeweils als Prologstruktur enthaelt.

Hierzu ein Beispiel:

```
append(nil,L,L).
append(cons(T,L1),L2,cons(T,L3)) :- append(L1,L2,L3).
```

Diese Prozedur sei dem Interpreter bereits eingegeben worden. Durch den Aufruf:

```
?- functor(Kopf,append,3),
   findall(klausel(Kopf,Rumpf),clause(Kopf,Rumpf),Prozedur).
```

liefert Prolog:

```
Prozedur = '.'(klausel(append(nil,_1,_1), true),
               '.'(klausel(append(cons(_2,_3),_4,cons(_2,_5)),
                           append(_3,_4,_5)), nil)).
```

(Anm. Der Interpreter bewahrt den Namen von Variablen nicht auf, sondern ersetzt sie durch Zahlen, wobei Variable mit denselben Namen auch dieselben Zahlen zugeordnet bekommen)

6.4. Die semantische Analyse

Dank der sehr einfachen und kompakten Semantik von Prolog, erweist sich die semantische Analyse einer Prologklausel als sehr einfach. Sie kann auf folgende Punkte beschraenkt werden:

a) Der Kopf einer Klausel muss eine Struktur, bzw ein Atom sein. (er darf weder eine Variable noch eine Integerzahl sein).

b) Die Praedikate im Rumpf der Klausel duerfen keine Integerzahlen sein. (Sollte ein Praedikat eine Variable sein, so ist es durch das Praedikat 'call(Variable)' zu ersetzen).

6.5. Die Behandlung von geschachtelten Strukturen.

Ein Grundgedanke in der Bearbeitung von Strukturen liegt darin, seine Argumente sequentiell zu bearbeiten. Dies ist jedoch nur moeglich wenn keines der Argumente einer Struktur wieder eine Struktur ist (dies wuerde ein rekursives Bearbeiten der Argumente erfordern). Um Substrukturen (geschachtelte Strukturen) aufzuloesen, gibt es ein einfaches Verfahren; die Substruktur wird durch eine Variable ersetzt, welche an die Substruktur gebunden wird.

Warren schlaegt vor, Substrukturen im Kopf und im Rumpf einer Klausel verschieden aufzuloesen. Im Rump einer Klausel wird zuerst eine Struktur selbst behandelt und hierauf

erst die ihr eingelagerten Substrukturen.

Im Rumpf einer Klausel wird umgekehrt vorgegangen. Zuerst werden jene Substrukturen behandelt, die in der tiefsten Schachtelungsebene liegen, dann erst ihre unterlagerten Strukturen. Diese Vorgangsweise hat folgenden Grund:

> Im Kopf einer Klausel ist es nicht sinnvoll zuerst jene Substrukturen zu behandeln, die in der tiefsten Schachtelungsebene liegen, da dadurch der uebergebene Parameter bis zu dieser Schachtelungsebene aufgeloest werden muesste (um das Unifizieren durchfuehren zu koennen), was sich nicht nur als sehr ineffizient erweist, sondern beim Erzeugen von neuen Strukturen gar nicht moeglich waere. (Eine noch nicht existente Struktur kann nicht aufgeloest werden). Wuerde im Rumpf einer Klausel jedoch ebenso vorgegangen, wie in dessen Kopf, so muesste Warrens Instruktionssatz erweitert werden, da keine Instruktion vorgesehen ist, um im Rumpf einer Klausel eine Struktur mit einer definierten Variable zu unifizieren.

Beim Entwurf des Zwischencodes fuer den zweiten Uebersetzer wurde dieses Konzept wie folgt veraendert:

Strukturen im Rumpf einer Klausel werden genauso aufgeloest wie geschachtelte Strukture im Kopf einer Klausel. Weiters wurde hierfuer ein neuer Zwischencode definiert. Da im Rumpf einer Klausel Strukturen am Heap aufgebaut werden, kann bereits zur Uebersetzungszeit ermittelt werden, wo die

einzelnen Substrukturen relativ zur Spitze des Heap stehen werden. Dieses Wissen nuetzt der Zwischencode aus, indem eine Instruktion eingefuehrt wird, welche fuer geschachtelte Strukturen einen Verweis auf jene Speicherzelle erstellt, bei der die Substruktur aufgebaut werden wird. Weiters wird die Behandlung des letzten Arguments einer Struktur optimiert, falls es sich dabei um eine Substruktur handelt. In diesem Fall steht die Substruktur unmittelbar nach der unterlagerten Struktur, was durch einen eigenen Befehl beruecksichtigt wird. (tail-Rekursion-Optimierung).

6.6. Speicherplatzvergabe an permanente Variable

Die permanenten Variablen erhalten im Environment einer Klausel Platz zugewiesen. Eine permanente Variable wird nach ihrem letzten Auftreten in einer Klausel gesichert und hierauf aus dem Environment entfernt. Dadurch kann zur Laufzeit das Environment getrimmt (kontrahieren) werden, was bei deterministischen Prozeduren eine sehr hohe Speichereffizienz garantiert.

Damit die Freigabe einer permanenten Variable jedoch Auswirkungen auf den Speicherplatzverbrauch hat, muss sie zum Zeitpunkt ihrer Freigabe an der Spitze des Environments stehen. Um diese Bedingung zu gewaehrleisten, genuegt es die Variablen entsprechend ihrem letzten Auftreten zu initialisieren. (jene Variable, die als letzte in einer Klausel auftritt, erhaelt den Index 1, die vorletzte den Index 2 usw.).

Ein Beispiel:

```
binaer_baum(T,baum(T,Info,Links,Rechts),Info).
binaer_baum(T,baum(T1,_,Links,Rechts),Info) :-
    T < T1,
    binaer_baum(T,Links,Info).
binaer_baum(T,baum(T1,_,Links,Rechts),Info) :-
    T > T1,
    binaer_baum(T,Rechts,Info).
```

Die Indizierung der Variablen sollte wie folgt durchgefuehrt werden (Xi sind temporaere Variable mit Index i, Yi sind permanente Variable mit Index i):

```
binaer_baum(X1,baum(X1,X2,_,_),X2).
binaer_baum(Y3,baum(X1,_,Y2,_),Y1) :-
    X1 < Y3,
    binaer_baum(Y3,Y2,Y1).
binaer_baum(Y3,baum(X1,_,_,Y2),Y1) :-
    X1 > Y3,
    binaer_baum(Y3,Y2,Y1).
```

6.7. Speicherplatzvergabe an temporaere Variable

Temporaere Variable besetzen denselben Speicherbereich, wie die Prozedurparameter. Dies bedeutet Ai belegt denselben Speicherplatz wie Xi. Aus dieser Tatsache lassen sich einige Optimierungen fuer die Vergabe von Indizes durchfuehren. Die Instruktionen

put_value Ai, Xi

und

get_variable Ai, Xi

entsprechen 'nop'-Instruktionen, d.h. sie benoetigen keine Operationsausfuehrung, da es sich jeweils um eine Eigenzuweisung (Ai := Ai) handelt. Beim Indizieren von temporaeren Variablen muss also versucht werden, X-Register und A-Register zu ueberlappen, ohne dabei die Konsistenz der Register zu gefaehrden (indem z.B. Xi mit einem Wert versehen wird, obwohl Ai noch nicht verarbeitet wurde), dass bei den Instruktionen 'get_variable' und 'put_value' moeglichst das A-Register und das X-Register denselben Speicherplatz belegen.

Ein Beispiel:

```
append(nil,L,L).
append(cons(T,L1),L2,cons(T,L3)) :-
   append(L1,L2,L3).
```

Die Indizierung der Variablen sollte wie folgt durchgefuehrt werden (Xi sind temporaere Variable mit Index i):

```
append(nil,X2,X3).
   get_constant nil, A1
   get_variable X2, A2      ==  'nop'
   get_value X2, A3

append(cons(X4,X1),X2,cons(X4,X3)) :-
   append(X1,X2,X3).
```

```
get_structure '.'/2, A1
unify_variable X4
unify_varialbe X1
get_variable X2, A2     == 'nop'
get_structure '.'/2, A3
unify_value X4
unify_variable X3
put_value X1, A1        == 'nop'
put_value X2, A2        == 'nop'
put_value X3, A3        == 'nop'
execute append/3
```

6.8. Indizierung von Klauseln

Anmerkung:

'In diesem Abschnitt steht 'Argument' stets fuer 'Argument des Kopfes einer Klausel'

Klauseln werden nach dem ersten Argument ihres Kopfs indiziert, wodurch ein schnellerer Zugriff auf jene Klausel ermoeglicht werden soll, die ein geeigneter Kandidat fuer die uebergebenen Parameter zu sein scheint. Bei der Indizierung muss jedoch beachtet werden, dass die Reihenfolge der Abfrage der Klauseln erhalten bleibt, da diese Eigenschaft einen wesentlichen Einfluss auf die exakte Ausfuehrung eines Prologprogrammes haben kann. Ist der erste Parameter einer Prozedur eine Variable so muss diese Klausel auf

jeden Fall getestet werden. Ist er jedoch eine Konstante bzw. eine Struktur, so werden mittels einer Hashtabelle jene Klauseln nicht mehr untersucht, die sicher beim Unifizieren des ersten Parameters mit dem ersten Argument ein 'fail' liefern wuerden.

6.8.1. Der Indizierungs-Algorithmus

Die einzelnen Klauseln werden in Teilbereiche unterteilt. Hierbei kommt folgender Algorithmus zur Anwendung. Die Klauseln seien ihrem Auftreten entsprechend von 1 bis n durchnummeriert.

1) I sei 1. Gehe zu Punkt 2.

2)

2a) Falls das erste Argument der i-ten Klausel eine Variable ist, so gilt diese Klausel als Teilbereich; I wird um 1 erhoeht; man gehe weiter bei Punkt 2.

2b) Fasse alle restlichen Klauseln zu einem Teilbereich zusammen, falls keine Klausel als erstes Argument eine Variable besitzt;

2c) Die Klauseln i bis (j-1) werden zu einem Teilbereich zusammen, wenn gilt:
j sei das kleinste j fuer das gilt, i < j <= n und die j-te Klausel besitzt als erstes Argument eine Variable.

i erhaelt den Wert von j; man gehe weiter bei Punkt 2.

Die durch den oben genannten Algorithmus erhaltenen Teilbereiche werden mit folgenden Instruktionen (die ein Backtracking ermoeglichen) logisch verkettet:

1) existiert nur ein Teilbereiche so sind keine Instruktionen fuer Backtracking noetig.

2) existieren 2 oder mehrere Teilbereiche, so steht vor dem ersten Teilbereich, die Instruktion 'try_me_else Label', vor dem letzten Teilbereich die Instruktion 'trust_me_else fail' und vor allen anderen Teilbereichen 'retry_me_else Label'; wobei 'Label' immer eine Referenz auf den naechsten Teilbereich darstellt.

Innerhalb der einzelnen Teilbereiche wird nochmals ein Mechanismus fuer Backtracking vorgesehen, der Zugriff auf die einzelnen Klauseln wird jedoch noch zusaetzlich durch einen hash-Algorithmus unterstuetzt. Die Backtracking-Instruktionen werden in gleicher Weise wie fuer die Teilbereiche vergeben; nur verknuepfen sie nun Klauseln statt Teilbereichen.

Sollte ein Teilbereich aus mehr als einer Klausel bestehen so werden ausserdem Instruktionen generiert, die es erlauben bei der Programmausfuehrung ueber eine Hashtabelle geeignete Klauseln zu finden. Dieses Verfahren wird durch

die Instruktion 'switch_on_type Var_ref,Konst_ref,Strukt_ref' zu Beginn einer Section eingeleitet, wobei 'Var_ref', 'Konst_ref' und 'Strukt_ref' als Verweise auf Klauseln dienen. 'Var_ref' verweist stehts auf die Backtracking-Instruktion der ersten Klausel.
Falls ein Teilbereich nur eine Klausel besitzt, die als erstes Argument eine Konstante (Struktur) hat, so zeigt 'Konst_ref' ('Strukt_ref') direkt auf diese Klausel, andernfalls werden Teilmengen gebildet, welche alle Klauseln, die als erstes Argument dieselbe Konstante (Struktur) besitzen zusammen\%fassen. 'Konst_ref' ('Struct_ref') verweist auf die Instruktion 'switch_on_constant Hashtabelle' ('switch_on_structure Hashtabelle'). Jede Teilmenge erhaelt nun einen Verweis in der Hashtabelle zugeordnet, wobei dieser Verweis, falls eine Teilmenge nur eine Klausel enthaelt, direkt auf diese Klausel zeigt, und andernfalls auf einen Block von Instruktionen zeigt, welche ein Backtracking zu den einzelnen Klauseln einer Teilmenge ermoeglichen. Der ebengenannte Block hat folgendes Format:

```
try klausel_i
retry klausel_j
retry klausel_k
trust klausel_l
```

Hierbei sind klause_n Verweise auf die n-te Klausel, wobei gilt i < j < ... < k < l.

Ein Beispiel:

```
klausel(konstant1, ...
klausel(konstant2, ...
klausel(struktur(...), ...
klausel(konstant1, ...
klausel(Var, ...
klausel(structure1(...), ...
klausel(structure2(...), ...
```

Diese Klauseln werden mit folgenden Backtracking und hash-Instruktionen ergaenzt:

```
        try_me_else S2
        switch_on_type T1,K1,K13
K1:     switch_on_constant (konstant1: C1,
                            konstant2: K12)
C1:     try K11
        trust K14

T1:     try_me_else T2
K11:    klausel(konstant1, ...
T2:     retry_me_else T3
K12:    klausel(konstant2, ...
T3:     retry_me_else T4
K13:    klausel(struktur(...), ...
T4:     trust_me_else fail
K14:    klausel(konstant1, ...

S2:     retry_me_else S3
K15:    klausel(Var, ...
```

```
S3:   trust_me_else fail

      switch_on_type T6, fail, St1

St1:  switch_on_structure (structure1/N: K16,

                           structure2/N: K17)

T6:   try_me_else T7

K16:  klausel(structure1(...), ...

T7:   trust_me_else T8

K17:  klausel(structure2(...), ...
```

7. Optimierung der Prologmaschine

Nachfolgende Optimierungsvorschlaege sind in der derzeitigen Implementierung nur teilweise vorgesehen. Sie sollen jedoch als Anstoss fuer kuenftige Imlpementierungen dienen.

a) Klauseln der Art f(X, ...) :- typ(X), ... (typ = var, nonvar, integer, atom, atomic, struct, functor, =..) muessen erkannt werden. Die 'switch_on_term' Instruktion kann diese zusaetzliche Information dazu verwenden, um ein praeziseres Auswahlkriterium von Klauseln zu erstellen.

b) Temporaere Variable sind, wenn moeglich, in Hardware-Registern zu halten. Dies ist daher sinnvoll, da die Lebensdauer einer temporaeren Variable beim Aufruf einer Prozedur erlischt. Um die Hardware-Register moeglichst effizient auszunuetzen muss das Register sofort nach dem letzten Benutzen der Variable freigegeben werden.

c) Da das erste Praedikat im Rumpf einer Klausel optimiert behandelt werden kann, (z.b. koennen Variable, die nur im Kopf und im ersten Praedikat auftreten noch als temporaer klassifiziert werden), sind builtin Preadikate wie !, var, atom usw. nicht als 'normale' Praedikate zu zaehlen (sie koennen direkt in Instruktionen umgewandelt werden).

d) Falls gewaehrleistet wird, dass die A-Register waehrend dem Unifizieren des Kopfes einer Klausel nicht veraendert werden, koennen Klauseln der Art f(...) :- !, ... optimiert werden.

Vor dem Unifizieren des Kopfs muessen nur der Heap pointer und der Trail pointer aufbewahrt werden, die bei einem 'fail' waehrend des Unifizierens zurueckzusetzen sind.

Weiters koennen choicepoints erst unmittelbar vor der Behandlung des ersten Praedikates vervollstaendigt werden, was eine schnellere Fehlerbehandlung ermoeglicht.

e) Durch Umstellen von put-Instruktionen koennen die Hardwareregister besser ausgenutzt werden. Z.B. kann eine put_void Instruktion an das Ende der put-instruktionen gesetzt werden, wodurch das entsprechende Register bis zu diesem Zeitpunkt frei benutzbar bleibt. (Auch put_constant sollte ans Ende der put-Instruktionen gestellt werden).

f) Durch eine staerkere Einschraenkung der Wirkungsweise der Instruktion 'unify_local_value' koennte der Heap weniger belastet werden. Die Einschraenkung waere:
Eine Variable wird nur dann auf den Heap gerettet, falls sie eine Referenz auf eine Variable enthaelt, welche vor dem naechsten Prozeduraufruf freigegeben wird.

g) Der Benutzer koennte Optimierungen erleichtern, indem er durch mode-Deklarationen angibt, ob bestimmte Parameter einer Klause immer definiert sind, bzw. immer undefiniert sind. Dadurch koennten manche 'unify_local_value' Instruktionen in die schneller exekutierbaren 'unify_value' Instruktionen umgewandelt werden.

h) Wird die Prologmaschine durch einen Interpreter emuliert, so empfiehlt es sich, den Instruktionssatz wie folgt zu erweitern:
Da in Prolog der Datentyp 'Liste' sehr haeufig ist, sind die Instruktionen 'get_structure '.'/2, Ai' und 'unify_constant nil', 'get_constant nil, Ai', 'put_constant nil, Ai' durch 'get_list Ai', 'unify_nil', 'get_nil Ai' bzw. 'put_nil Ai' zu ersetzen.
Instruktionen mit haeufig auftretenden Registerbelegungen koennen ebenfalls als neue Instruktion eingefuehrt werden. Z.B:
'get_nil_1', fuer ''get_nil A1' usw.
Diese Massnahmen helfen nicht nur den Code zu reduzieren, sondern erhoehen auch die Geschwindigkeit eines Interpreters.

i) Durch intelligentes Backtracking kann die Laufzeit sehr verkuerzt werden. Beim intelligenten Backtracking wird bei einem 'fail' die Programmausfuehrung nicht automatisch beim letzten choicepoint wieder aufgenommen, sondern durch zusaetzliche Information geht das Back-

tracking bis zu jenem choicepoint zurueck, bei dem jene Bindung, die das 'fail' verursachte aufgeloest, und mit einem neuen Wert gebunden werden kann.
Ein Beispiel:

```
a :- b(X), c(Y,Z), d(X).
```

Hier wuerde bei einem fail im Praedikat 'd', nicht das Praedikat 'c' reaktiviert, (ausser X waere an Y bzw. Z gebunden), sondern das Praedikat 'b' wuerde als neuer choicepoint dienen. (Zum Thema 'intelligentes Backtracking' seinen besonders Artikel von M. Bruynooghe und L.M. Pereira empfohlen).

8. Beispiele

```
      qsort(L,R) :- qsort(L,R,nil).

qsort(nil,R,R).
qsort(cons(X,L),R0,R) :-
   split(L,X,L1,L2),
   qsort(L1,R0,cons(X,R1)),
   qsort(L2,R1,R).

split(nil,X,nil,nil).
split(cons(Y,L),X,cons(Y,L1),L2) :-
   less_than(Y,X),
   split(L,X,L1,L2).
split(cons(Y,L),X,L1,cons(Y,L2)) :-
   greater_or_equal(Y,X),
   split(L,X,L1,L2).


qsort/2:                          % qsort(
                                  % L,
                                  % R) :-
                                  % qsort(L,
                                  % R,
      put_constant nil, A3        % nil
      execute  qsort/3            % ).

qsort/3:
```

```
        switch_on_type T1,K11,K12
T1:     try_me_else T2              % qsort(
K11:    get_constant nil, A1        % nil,
                                    % R,
        get_value X2, A3            % R
        proceed                     % ).

T2:     trust_me_else fail          % qsort(
K12:    allocate
        get_structure '.'/2, A1     % cons(
        unify_variable Y4           % X,
        unify_variable X1           % L),
        get_variable Y5, A2         % R0,
        get_variable Y1, A3         % R) :-
                                    % split(L,
        put_value Y4, A2            % X,
        put_variable Y6, A3         % L1,
        put_variable Y3, A4         % Y2
        call split/4, 6             % ),
        put_unsafe_value Y6, A1     % qsort(L1,
        put_value Y5, A2            % R0,
        put_structure '.'/2, A3     % cons(
        unify_value Y4              % X,
        unify_variable Y2           % R1)
        call qsort/3, 3             % ),
        put_unsafe_value Y3, A1     % qsort(L2,
        put_value Y2, A2            % R1,
        put_value Y3, A3            % R
```

```
        deallocate
        execute qsort/3          % ).


split/4:
        switch_on_type T1,K11,St1
        switch_on_structure ('.'/2: B1)
B1:     try K12
        trust K13

T1:     try_me_else T2           % switch(
K11:    get_constant nil, A1     % nil,
                                 % X,
        get_constant nil, A3     % nil,
        get_constant nil, A4     % nil
        proceed                  % ).

T2:     retry_me_else T3         % switch(
        allocate
K12:    get_structure '.'/2, A1  % cons(
        unify_variable X1        % Y,
        unify_variable Y4        % L),
        get_variable Y3, A2      % X,
        get_structure '.'/2, A3  % cons(
        unify_value X1           % Y,
        unify_variable Y2        % L1),
        get_variable Y1          % L2) :-
                                 % less_than(Y,
        put_value Y3             % X
```

```
        call less_than/2, 4         % ),
        put_value Y4, A1            % split(L,
        put_value Y3, A2            % X,
        put_value Y2, A3            % L1,
        put_value Y1, A4            % L2
        deallocate
        execute split/4             %).

T3:     trust_me_else fail          % switch(
        allocate
K13:    get_structure '.'/2, A1     % cons(
        unify_variable X1           % Y,
        unify_variable Y4           % L),
        get_variable Y3, A2         % X,
        get_variable Y2             % L1,
        get_structure '.'/2, A4     % cons(
        unify_value X1              % Y,
        unify_variable Y1           % L2)) :-
                                    % greater_or_equal(Y,
        put_value Y3                % X
        call greater_or_equal/2, 4  % ),
        put_value Y4, A1            % split(L,
        put_value Y3, A2            % X,
        put_value Y2, A3            % L1,
        put_value Y1, A4            % L2
        deallocate
        execute split/4             %).
```

```
compile(variable(No,loc(X,Y,ref(R,S)),safe),C) :-
    analyse(term(X,variable(No,ref(S),C),C).

      get_structure variable/3, A1   % compile(variable(
      unify_variable X3              % No
      unify_variable X1              % V == loc(X,Y,ref(R,S))
      unify_constant safe            % safe),
      get_structure loc/3, X1        % loc(
      unify_variable X4              % X,
      unify_void 1                   % Y,
      unify_last_structure ref/2     % ref(
      unify_void 1                   % R,
      unify_variable X5              % S)
                                     % C) :-
      put_structure term/2, A1       % analyse(term(
      match_value X4                 % X,
      match_last_structure variable/3 % variable(
      match_value X3                 % No,
      match_referenz 2               % ref(S),
      match_local_value X2           % C),
      match_structure ref/1          % ref(
      match_value X5                 % S),
                                     % C
      execute analyse/2              % ).
```

```
arbeiter(maier,franz).
arbeiter(krall,andreas).
arbeiter(maier,johanna).
arbeiter(X,Y) :-
   praktikant(X,Y).
arbeiter(X,Y) :-
   freier_mitarb(X,Y).
arbeiter(koordinator(klein),hermine).
arbeiter(boese,herbert).
arbeiter(maier,josef).
arbeiter(vorstand(ambach),irene).
arbeiter(vorstand(morandell),anna).
```

```
arbeiter/2:
S1:   try_me_else S2
      switch_on_type T1,K1,fail
K1:   switch_on_constant (maier: B1,
                          krall: K12)
B1:   try K11
      trust K13
T1:   try_me_else T2          % arbeiter(
K11:  get_constant maier      % maier,
      get_constant franz      % franz
      proceed                 % ).
T2:   retry_me_else T3        % arbeiter(
K12:  get_constant krall      % krall,
      get_constant andreas    % andreas
```

```
        proceed                 % ).
T3:     trust_me_else fail      % arbeiter(
K13:    get_constant maier      % maier,
        get_constant johanna    % johanna
        proceed                 % ).

S2:     retry_me_else S3        % arbeiter(X,Y) :-
        execute praktikant/2    % praktikant(X,Y).

S3:     retry_me_else S4        % arbeiter(X,Y) :-
        execute freier_mitarb/2 % freier_mitarb(X,Y).

S4:     trust_me_else fail
        switch_on_type T6,K2,St1
K2:     switch_on_constant  (boese: K17,
                             maier: K18)
St1:    switch_on_structure (koordinator/2: K16
                             vorstand/2: B2)
B2:     try K19
        trust K110

T6:     try_me_else T7          % arbeiter(
K16:    get_structure koordinator/1, A1  %koordinator(
        unify_constant klein    % klein),
        get_constant hermine    % hermine
        proceed                 % ).
T7:     retry_me_else T8        % arbeiter(
K17:    get_constant boese      % boese,
        get_constant herbert    % herbert
```

```
        proceed                     % ).
T8:     retry_me_else T9            % arbeiter(
K18:    get_constant maier          % maier,
        get_constant josef          % josef
        proceed                     % ).
T9:     retry_me_else T10           % arbeiter(
K19:    structure vorstand/1, A1    % vorstand(
        unify_constant ambach       % ambach),
        get_constant irene          % irene
        proceed                     % ).
T10:    trust_me_else fail          % arbeiter(
K110:   structure vorstand/1, A1    % vorstand(
        unify_constant morandell    % morandell),
        get_constant anna           % anna
        proceed                     % ).
```

9. Literaturliste

/BR82/ M. Brockhaus, F. Reichl: Uebersetzerbau (Skriptum)

/IF84/ Fa. InterFace Computer (Muenchen): IF/Prolog - Manual, 1984

/CM81/ W.F. Clocksin, C.S. Mellish: Programming in Prolog, Springer Verlag, 1981

/MF83/ T.Moto-oka, K.Fuchi: new generation computing (an international Journal on fifth generation computers) vol. 1 no. 1, Springer Verlag, 1983

/MF84/ T.Moto-oka, K.Fuchi: new generation computing (an international Journal on fifth generation computers) vol. 2 no. 1, Springer Verlag, 1984

/CT82/ K.L.Clark, S.A.Taernlund: Logic Programming, academic press, 1982

/CA84/ J.A. Campbell: implementations of Prolog, Ellis Horwood limited, 1984

/W177/ David H.D. Warren: Implementing Prolog Volume 1, D.A.I research report no 39, 1977

/W277/ David H.D. Warren: Implementing Prolog Volume 2, D.A.I research report no 40, 1977

/WA83/ David H.D. Warren: an abstract prolog instruction set, SRI project 4776, 1983

Inhaltsverzeichnis